Minispec Reference Guide

Version 1.0, September 2019

1 Introduction

This document is the primary reference for the Minispec hardware description language. It describes the syntax and semantics of the language in detail, and illustrates their use.

This reference is not intended as a first introduction to Minispec, and assumes some familiarity with the language. We recommend the Minispec tutorials as an introduction to the language.

How to use this reference: This reference does not assume you will be reading it sequentially. Use it to answer specific questions or to improve your knowledge of particular aspects of Minispec.

Each section presents a particular aspect of the language, with sections roughly laid out bottom-up: the first sections present the basic elements of the language (tokens, types, expressions, etc.), while the latter sections present the more complex elements (functions, modules, etc.), which build on the simpler ones.

Each section presents code examples in `gray insets`. In syntax descriptions, items enclosed in [red brackets] are optional.

Acknowledgments: Minispec is very closely related to Bluespec SystemVerilog (BSV): it shares much of its syntax with BSV and the Minispec compiler internally translates Minispec to BSV. Minispec simply would not exist without BSV. Nonetheless, using Minispec requires no knowledge of BSV, and so this reference is *self-contained*: it presents Minispec on its own, without any further reference to BSV. Minispec from BSV recounts the differences between both languages.

In writing this reference, we took inspiration from the excellent language references of BSV, Rust, and Python.

2 Lexical Structure

This section describes the types of tokens or lexemes in Minispec, i.e., its most basic syntax elements.

2.1 Whitespace and comments

Spaces, tabs, and newlines all constitute whitespace. Minispec is a *free-form language*: whitespace serves only to separate tokens and has no other significance.

Comments are treated as whitespace, and can either be one-line comments starting with `//`, or multiline comments delimited by `/*` and `*/`. Comments do not nest.

```
// Example one-line comment
/* You can write comments
   across multiple lines */
```

2.2 Identifiers and capitalization

Identifiers, also called names, are non-empty strings where:

- The first character is a lowercase letter, an uppercase letter, or a dollar sign (`$`).
- The remaining characters are alphanumeric or underscore (_).

Capitalization is important in Minispec. Identifiers for type names, module names, and enum labels *must begin with an uppercase letter*. Every other identifier (i.e., those for variables, function names, method names, rule names, input names, and instance names) *must begin with a lowercase letter*.

Identifiers whose first character is $ are reserved for *system functions* (see Section 12).

```
// Uppercase identifiers
Bool
FooType

// Lowercase identifiers
fooVar
barFunction

// Dollar-sign identifiers
$display
$finish
```

2.3 Keywords

The following words are reserved *Minispec keywords*, and cannot be used as normal identifiers:

```
type        function      method      input     begin     for          let
typedef     endfunction   endmethod   default   end       return
struct      module        rule        case      if        import
enum        endmodule     endrule     endcase   else      bsvimport
```

In addition, because Minispec compiles to BSV and Verilog, the Minispec compiler currently forbids using keywords from these languages as identifiers. The compiler will produce an error if any of these keywords is used.

2.4 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.

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.

```
// 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

// _ can separate digits
16'b1010_0110_1101_0010
```

```
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
```

Unsized literals must be used in expressions involving `Integer` variables, as `Integer` is an unsized, compile-time-only type (see Section 6.3).

2.5 String literals

String literals are enclosed in double quotes (`"..."`) and must be written on a single line. String literals admit special characters using the following escape sequences:

```
"This is a string"
"String with a\nline break"
```

Character	newline	tab	backslash	double quote
Escape sequence	`\n`	`\t`	`\\`	`\"`

2.6 Bool and Maybe literals

`True` and `False` are literals of type `Bool` (Section 6.1). `Invalid` and `Valid` are literals of type `Maybe` (Section 6.5). Like keywords, these literals cannot be used as normal identifiers.

2.7 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 and can be used in place of literals. They help the compiler produce better circuits (see Section 8.1 for examples).

3 Overview of Types

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.

Built-in types: Minispec provides five basic, built-in types:

- `Bool` represents a Boolean value, which can be either `True` or `False`.
- `Bit#(n)` represents an n-bit value.
- `Vector#(n, T)` represents a collection of n values of type T.
- `Maybe#(T)` represents an optional value of type T.
- `Integer` represents an integer value with an unbounded number of bits.

Section 6 describes and illustrates the use of these types in detail. All types except `Integer` are ultimately represented as a collection of bits in hardware, and can be converted to and from `Bit#(n)` (Section 6.2). `Integer` is special: it cannot be synthesized into hardware and can only be used at compile time.

User-defined types: Minispec supports three kinds of user-defined types:

- *Type synonyms* allow giving a different name to a type.
- *Structs* represent collections of values.
- *Enumerations* or enums allow defining a set of unique symbolic constants, also called labels.

Section 7 describes user-defined types in detail.

Parametric types: Types of the form `Type#(`ρ_1`,...,`ρ_n`)` are called parametric types (also known as polymorphic types or generics in other languages). Parametric types take one or more parameters (ρ_i), which can be either `Integer` values or other types. These parameters must be known at compile time, and make the type concrete. Several of the built-in types are parametric, as shown in the examples.

```
// A 16-bit value
Bit#(16) halfWord;

// A Vector of 8 Bools
Vector#(8, Bool) boolVec;

// A Vector of 16 Bit#(32)s
Vector#(16,Bit#(32)) wordVec;
```

Beyond parametric types, Minispec also supports parametric functions, modules, structs, and type synonyms using a consistent syntax and semantics, which Section 10 describes in detail.

Type conversions are explicit, except for Integers: 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.

```
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
```

`Integer` values are the single exception to this rule: `Integer` values can be assigned to `Bit#(n)` variables without explicit conversion. In fact, `Integer`s can be used anywhere a unsized number literal would work, as shown in the examples.

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

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.

```
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
```

4 Expressions

Expressions are a combination of variables, literals, operators, and function calls that are *evaluated* to produce a value. Expressions appear in the right-hand side of assignments, as arguments to function calls, and generally anywhere a value is needed.

This section describes the different types of expressions and their syntax, but *does not detail the semantics of most expression types*. For example, the operators subsection below explains the syntax and precedence of operators, but does not detail what each one *does*. This is because these semantics are type-specific, so they are better explained later (e.g., in Section 6 for operators).

Simple and complex expressions: Expressions may be as simple as a single variable or literal, may involve a single operator or function call (e.g., `a && b` or `foo(x)`), or may consist of multiple nested *subexpressions* (e.g., `foo(a && b) || c`).

Parentheses, precedence, and associativity: Complex expressions with multiple operators can use parentheses to explicitly specify the evaluation order of the multiple operations (e.g., `(a + b) * c` or `a + (b * c)`).

When parentheses are not used, evaluation order is determined using the *precedence* and *associativity* rules of the different operators. Each operator has a precedence, and higher-precedence operators are evaluated first. For example, `a + b * c` is equivalent to `a + (b * c)` because `*` has higher precedence than `+`.

Among operators with the same precedence, associativity determines evaluation order. Almost all operators are *left-associative*, so the expression is evaluated left-to-right. For example, `x << 4 << 2` is equivalent to `(x << 4) << 2` (i.e., `x << 6`), and *not* `x << (4 << 2)` (which would be `x << 16`). The only right-associative operator is the ternary operator (Section 4.2).

Though these rules match those of most programming languages and should feel natural, we recommend using parentheses when the evaluation order is not immediately obvious.

No side effects: Expressions in Minispec *never* have side effects, i.e., evaluating an expression never modifies any variable or register. This means that independent subexpressions in a complex expression can be evaluated in any order (or in parallel). For example, in `foo(a && b) || foo(b)`, both calls to `foo` can be evaluated in any order (function calls have no side effects).

4.1 Unary and binary operators

The tables below and to the right detail Minispec's unary and binary operators, respectively. Unary operators always have higher precedence than binary operators (so, for example, `!a && b` is equivalent to `(!a) && b`). Binary operators are shown in descending precedence order. Operators in the same row have the same precedence. All binary operators are left-associative.

Unary operators

Type	Operators	Names
Boolean	!	Boolean NOT
Logical	~	Bitwise invert
Arithmetic	+, -	Unary plus, negation
Reduction	&, |, ^	AND/OR/XOR bit reduction

Binary operators in precedence order

Type	Operators	Names
Arithmetic	**	Exponentiation
	*, /, %	Multiplication, division, modulus
	+, -	Addition, subtraction
	<<, >>	Left shift, right shift
Relational	<, >,<=, >=	Less/greater-than/or-equal
Equality	==, !=	Equality, inequality
Logical	&	Bitwise AND
	^	Bitwise XOR
	^~, ~^	Bitwise XNOR
	|	Bitwise OR
Boolean	&&	Boolean AND
	||	Boolean OR

Most operators admit operands of particular types. Specifically:

- Boolean operators apply only to values of type `Bool`.
- Arithmetic, relational, bitwise logical, and bit reduction operators apply only to values of types `Bit#(n)` or `Integer`.
- Equality operators apply to values of any type, including user-defined types.

4.2 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 Boolean 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`.

The conditional operator is right-associative (in fact, it is the only right-associative operator). Therefore, `a? x : b? y : z` is equivalent to `a? x : (b? y : z)`, and *not* `(a? x : b)? y : z`.

The conditional operator has the lowest precedence of all other operators. Therefore, `a? x : y + z` is equivalent to `a? x : (y + z)`, and *not* `(a? x : y) + z`.

Case expression: The case expression allows selecting among multiple values depending on a value. Its syntax is:

```
case (compExpr)
  value1 : expr1;
  value2 : expr2;
        ...
[ default : defaultExpr; ]
endcase
```

where `compExpr`, $\texttt{expr}_i$, and `defaultExpr` are expressions, and $\texttt{value}_i$ are different values of the same type as `compExpr`. If `compExpr`'s value matches one of the values $\texttt{value}_i$, the case expression evaluates to its corresponding expression $\texttt{expr}_i$. Otherwise, the case expression evaluates to `defaultExpr`.

The case expression must match against something—it must always evaluate to a value. Therefore, the `default` item is optional only when the case expression enumerates all possible values of `compExpr`. A case expression that does not enumerate all possible values and does not have a default item will produce a compiler error.

```
Bit#(2) v = 2'b10;

Bool isOdd = case (v)
  2'b01 : True;
  2'b11 : True;
  default : False;
endcase;

// Can omit default if we
// list all possibilities
Bit#(2) plusOne = case (v)
  0 : 1;
  1 : 2;
  2 : 3;
  3 : 0;
endcase;
```

4.3 Selection and concatenation expressions

Selection expressions use brackets (`[]`) to select a bit or range of bits from a `Bit#(n)` value, or an element from a `Vector#(n,T)`. Their syntax is `expr[ startExpr[:endExpr] ]`. Section 6.2 describes the semantics of selection expressions for `Bit#(n)`, and Section 6.4 describes the semantics for `Vector#(n,T)`.

Concatenation expressions use curly braces (`{}`) to concatenate multiple `Bit#(n)` values into a wider value. Their syntax is `{expr1, ..., exprN}`. Section 6.2 describes their semantics.

4.4 Struct creation expressions

The struct creation expression creates a struct value. Its syntax is `StructType{member1:expr1, ..., memberN:exprN}`. Section 7.2 describes structs and the semantics of struct creation.

4.5 Function and method calls

Functions are described in Section 8. The function call expression allows invoking a function. Its syntax is:

```
funcName[#(param1, ..., paramK)][(argExpr1, ..., argExprN)]
```

where `funcName` is the name of the function, $\texttt{param}_i$ are the function's parameters, and $\texttt{argExpr}_i$ are the function's arguments. Parameters must be specified if the function is parametric (see Section 10 for details and examples on parametric functions), and can't be specified otherwise. Arguments are optional because it's possible to have functions without arguments.

Methods allow modules to return values, and are described in Section 9. The method call expression has the following syntax:

```
submoduleName.methodName[(argExpr1, ..., argExprN)]
```

where `submoduleName` is the name of the submodule instance that implements the method, `methodName` is the name of the method, and $\texttt{argExpr}_i$ are the method's arguments. Methods cannot be parametric.

5 Statements

Statements, such as variable declarations or assignments, are the basic syntax unit of Minispec logic. For example, each function (Section 8) consists of a sequence of statements separated by semicolons. Each statement expresses an action to be carried out. Whereas expressions *always* evaluate to a value and *never* have side effects, statements *never* evaluate to a value and *often* have side effects. Semantically, statements appear to be evaluated in sequence, which makes code easy to understand. However, sequences of statements are synthesized into parallel combinational logic. Section 8.1 explains how this synthesis is done.

5.1 Variable statements

Variables are names for intermediate values.

Variable declaration and initialization: The basic variable declaration statement has the form:

```
VarType varName;
```

where `VarType` is the variable's type and `varName` is its name. Variables declared this way are *uninitialized*, and cannot be used until they are assigned a value.

Variables can be declared and initialized in the same statement:

```
VarType varName = initExpr;
```

where, `initExpr` is an expression compatible with type `VarType`.

When declaring and initializing a variable, the `let` keyword can be used in place of `VarType`:

```
let varName = initExpr;
```

In this case, the compiler will infer the variable's type from `initExpr`'s type.

Finally, multiple variables of the same type can be declared and optionally initialized in the same statement:

```
VarType varName1 = initExpr1, ..., varNameN = initExprN;
```

Variable assignment: Each assignment statement binds a variable to a value. Its syntax is `lValue = expr;`, where `lValue` can be *(i)* variable name, *(ii)* a struct member or submodule input (e.g., `structVarName.memberName`), *(iii)* a single bit or vector element (e.g., `varName[bitNum]`), *(iv)* a range of a bits from a `Bit#(n)` variable (e.g., `varName[hi:lo]`), or *(v)* a nested combination of any of the above.

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.

Variables of compound types (i.e., structs, vectors, and `Bit#(n)`) 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` or `?`), then reassign the individual bits. Structs and vectors do not suffer from this limitation.

```
// Variable declarations
//  Without initialization
Bit#(1) a;
//  With initialization
Bit#(2) b = 2'b11;
//  Using let
let c = 3'b011; // Bit#(3)
//  Multiple variables
Bool d = True, e = False;

// Assignments
a = 1'b1;
b[1] = 1'b0;
c[2:1] = b + 1;
typedef struct {
  Bool m; Bit#(2) n;
} ExStruct;
ExStruct s;
s.n[1] = 0;

// Lexical scoping
if (d) begin
  let x = !e;
  $display(x); // in scope
end
$display(x);
// Error, x out of scope

// Shadowing
if (d) begin
   let d = !e; // shadows
   $display(d);
end
$display(d); // OK, uses
             // outer d
```

Variables are lexically scoped: Like most languages, Minispec uses *lexical scoping*: a variable may only be used inside the block of code it is defined. For example, a variable declared within a begin-end block cannot be used outside the block.

Name clashes and variable shadowing: It is illegal to declare two or more variables with the same name in the same scope (i.e., block of code). Scopes can be nested, and it is allowed, *though not recommended*, to declare a variable in a nested scope with the same name as another variable in a surrounding scope. In this case, we say the variable in the inner scope *shadows* (i.e., hides) the variable in the outer scope. Shadowing can be confusing and error-prone, so the compiler emits warnings on every shadowed variable.

5.2 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`. Its syntax is:

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

where $\texttt{stmt}_\texttt{i}$ are statements. Each begin-end block initiates a new lexical context, which supports local variable declarations.

Note that *begin-end statements are not terminated by semicolons*, and placing a `;` after the `end` keyword will cause an error. This follows BSV and SystemVerilog/Verilog syntax. In general, keywords that start with `end` (e.g., `end`, `endfunction`, `endcase`, etc.) are not followed by `;`, while all other statements are terminated by `;`.

5.3 Control-flow statements

Minispec supports conditional statements and (restricted) loops. Though similar in syntax to those of other programming languages, these constructs synthesize to combinational logic. Here we explain the syntax and semantics of each statement, and Section 8.1 explains their synthesis to hardware.

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` is executed. Otherwise, if the optional else clause is present, `falseStmt` is executed.

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

Case statements have a similar syntax to case *expressions* (Section 4.2):

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

A case statement tests `compExpr` against the $\texttt{value}_i$ values, and on a match with $\texttt{value}_i$, $\texttt{stmt}_i$ is executed. If there are no matches and the optional default label is specified, `defaultStmt` is executed. 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 is executed.

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 iterations; `updExpr` is evaluated after each iteration to update `iVar`; and `stmt` is the statement executed on each iteration.

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*. This makes for loops implementable with combinational logic. To guarantee that the iteration count is known, the induction variable is always an `Integer`. For example, the loop in the example to the right is unrolled into:

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

Return statements are used to return a value from a function or method. Their syntax is simply `return expr;`. Section 8 explains where they can appear and provides examples.

5.4 Register-write statements

A register-write statement performs a write to a register module. Its syntax is `regName <= expr;`. Registers are the basic building block of modules and sequential logic, so Section 9 details the semantics and restrictions of register writes.

6 Built-In Types

This section describes the five basic built-in types that Minispec provides. It details the semantics of the operators each type supports and describes the built-in functions provided to work with these types.

6.1 Bool

Variables 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 (&&) and Boolean OR (`||`).

`Bool` values also support equality operations (==, `!=`). Note that, for Boolean values, `!=` is equivalent to Boolean XOR, and == is equivalent to Boolean XNOR.

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

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

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

6.2 Bit#(n)

`Bit#(n)` represents an `n`-bit value. The parameter `n` must be a non-negative `Integer`.

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.

Arithmetic and relational operations: `Bit#(n)` supports all arithmetic and relational operators. These operations treat `Bit#(n)` values as *unsigned integers*.

Bit reduction operations: `Bit#(n)` supports AND (&), OR (|), and XOR (^) reductions. Bit reductions are unary operations that take a `Bit#(n)` input and return a `Bit#(1)` output that results from reducing the bits of the input using the specified operation. For example, given `Bit#(4) a`, `&a` is equivalent to `a[3] & a[2] & a[1] & a[0]`.

```
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#(4) s = a + b; // 4'b1000
Bool geq = a >= b; // False

Bit#(1) any1 = |a; // 1'b1
Bit#(1) ev1s = ^a; // 1'b0
```

Converting to and from `Bit#(n)`: All types except `Integer` are ultimately represented as a collection of bits in hardware, and can be converted to and from `Bit#(n)`. The built-in function `pack` converts to `Bit#(n)`, and `unpack` converts from `Bit#(n)`, as shown below.

```
typedef enum Color {Red, Green, Blue};
Color c = Red;
Bit#(2) x = pack(c);      // 2'b00, the binary representation of Red
Color g = unpack(x + 1);  // Green, whose binary representation is 2'b01
Color u = unpack(x + 3);  // unspecified value, as no Color corresponds to 2'b11
```

Bit selection: Given a `Bit#(n)` variable `x`, `x[i]` selects the i^{th} 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 the value is $\sum_{i=0}^{n-1} 2^i x[i]$), but it is different from how vectors are indexed.

The selector `i` in `x[i]` can be a literal, an `Integer`, or a `Bit#(k)` variable. In the first two cases the bit selected is constant, which results in an efficient hardware implementation (as no logic gates are required, only wires). In the last case the bit selected is variable, and will require more hardware to implement.

Given a `Bit#(n)` variable `x`, `x[i:j]`, with `i` $\geq$ `j`, selects the range of bits of `x` starting at `x[i]` and ending at `x[j]`, both inclusive. When the difference between `i` and `j` is known at compile time, `x[i:j]` has type `Bit#(i-j+1)`. Otherwise, the bit-width of the selected slice is unknown at compile time, and on a size mismatch with the destination variable, the slice will be padded with zeros if the destination is wider, and the slice will be truncated from the left if the destination is narrower.

Bit concatenation: Given two or more `Bit#()` values $x_1, ..., x_k$, $\{x_1, ..., x_k\}$ concatenates these values. The result has type `Bit#(s)`, where `s` is the sum of the bit-widths of all concatenated elements.

Bit concatenation can take at most one unsized number literal, and will infer the width of that literal to match the widths of the concatenated value and the required type, as shown in the examples below.

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

```
Bit#(2) a = 2'b11;
Bit#(8) i = {a, 0};            // 8'b1100_0000, as 0 is inferred to be 6 bits
Bit#(8) j = {2'b11, 0, 1'b1};  // 8'b1100_0001, as 0 is inferred to be 5 bits
Bit#(8) k = {~0, ~a, a};       // 8'b1111_0011, as 0 is inferred to be 4 bits
Bit#(8) k = {0, 10};           // Error, can't deduce the sizes of two unsized literals
```

Truncation and extension: The built-in function `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. Finally, `signExtend` extends the argument by replicating its most significant bit to match the bit-width of a wider destination.

```
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
```

6.3 Integer

`Integer` represents an integer value (i.e., a negative or non-negative number) with an unbounded number of bits. `Integer` supports the same unary and binary operators as `Bit#(n)`. 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, and it is used exclusively at compile time: all `Integer` expressions must be evaluable by the compiler. Section 10 explains precisely how `Integer` values are evaluated at compile time.

6.4 Vector#(n,T)

`Vector#(n, T)` represents a vector or array of `n` elements of type `T`.

Element selection: Given a `Vector#(n,T)` variable `v`, `v[i]` selects the i^{th} bit of `v`. `v[i]` has type `T`.

The selector `i` in `v[i]` can be a literal, an `Integer`, or a `Bit#(k)` variable. As in bit selection (Section 6.2), only the last of the three cases results in dynamic selection, which requires more hardware.

Static and dynamic element selection have different guarantees with respect to out-of-bounds accesses. Static selectors are always checked by the compiler, so the compiler will flag any `v[i]` with `i >= n` as an out-of-bounds error. Dynamic selectors are not checked, so `v[i]` with `i >= n` will return an unspecified element.

Initialization: Vector variables need not be initialized when declared, and can be initialized element-by-element.

Modules (Section 9) can instantiate *Vectors of submodules*. In this case, the vector is initialized with the same syntax as submodule instantiation (Section 9.1): `Vector#(n,T) vectorName(submoduleArg1, ..., submoduleArgK);`. Submodule arguments are passed to all submodules.

Other vector functions: `Vector#(n,T)` is a BSV type. Appendix C.3 of the BSV reference describes several functions for working with vectors, which can also be used in Minispec code.

```
Vector#(4, Bool) lessThanTwo;
for (Integer i=0; i<4; i=i+1)
  lessThanTwo[i] = i < 2;

// Static selection
Bool a = lessThanTwo[3];
Integer i = 1;
Bool b = lessThanTwo[i+1];
// Error, out-of-bounds
Bool c = lessThanTwo[i+3];

// Dynamic selection
Bit#(3) j = 3'd1;
Bool d = lessThanTwo[j];
// Undefined, out-of-bounds
Bool e = lessThanTwo[j+3];

// Vector of submodules
module Example;
  // Vector of 4 8-bit regs,
  // all initialized to 0
  Vector#(4, Reg#(Bit#(8)))
    regVector(4'd0);
  [...]
endmodule
```

6.5 Maybe#(T)

`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 (Section 9), 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`.

```
// Creating Maybe#(T)'s
Maybe#(Bit#(1)) a = Valid(1);
Maybe#(Bit#(4)) b = Invalid;

// Validity checking
Bool aValid = isValid(a);

// Unpacking Maybe#(T)'s
let x = fromMaybe(0, a); // 1
let y = fromMaybe(4, b); // 4

// Common unpacking idiom:
// the if condition checks
// validity, so fromMaybe's
// arg is always valid and
// defaultValue is irrelevant
if (isValid(a)) begin
  let aVal = fromMaybe(?, a);
  [...]
end
```

7 User-Defined Types

7.1 Type synonyms

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

```
typedef Type NewType;
```

```
typedef Bit#(8) Byte;
Byte x = 8'd1;
```

where `Type` is any existing type, and `NewType` is the new type's name. Type synonyms are a convenience feature. The original type and its synonym can be used interchangeably, e.g., without any type conversions on assignments.

Type synonyms can be parametric; see Section 10 for details.

7.2 Structs

Structs are composite types (also known as product types): they represent a group of *members* of different types.

Definition: Struct definitions use the following syntax:

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

where `StructType` is the (new) type for the struct, $\texttt{member}_i$ are the (lowercase) names of its members, and $\texttt{Type}_i$ are the (uppercase) types of its members.

Structs can be parametric; see Section 10 for details.

Creation: The struct creation expression (Section 4.4) allows constructing a struct value. Its syntax is:

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

where `StructType` is the struct's type name, $\texttt{member}_i$ are the struct members' names, and $\texttt{expr}_i$ denote the values that the members should take.

Member access: Given value `s` of a struct type that has a member with name `m`, the expression `s.m` yields the value of member `m`. If `s` is a variable, then values can be assigned to its individual members as shown in the example.

```
// Definition
typedef struct {
  Byte red;
  Byte green;
  Byte blue;
} Pixel;

// Creation
Pixel cyan = Pixel{
  red : 0,
  green : 255,
  blue : 255
};

// Member access
Bit#(10) intensity =
  zeroExtend(cyan.red) +
  zeroExtend(cyan.green) +
  zeroExtend(cyan.blue);

// Assignment
Pixel white = cyan;
white.red = 255;
```

7.3 Enums

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 $\texttt{Label}_\texttt{i}$ 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 N possible labels as a $\lceil log_2 N \rceil$-bit value. With the syntax above, $\texttt{Label}_\texttt{i}$ will take 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 $\texttt{val}_\texttt{i}$ are the distinct numeric values of the labels.

```
// Enum definition
typedef enum {
  Ready, Busy, Error
} State;

// Usage example; assume
// that mod is a module
State s = mod.getState();
if (state == Ready)
  // Feed mod an input
else if (state == Error)
  // Halt system

// Enum def with label values
typedef enum {
 Red = 0, Blue = 2, Green = 1
} PixelChannel;
```

8 Functions and Combinational Logic

Functions implement combinational logic (also known as Boolean logic). Each function takes one or more *input arguments* and produces an *output result* that depends only on the values of the arguments, and does not depend on any other state.

Definition: Functions can be defined using the following syntax:

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

where `fname` is the function's name; `RetType` is the type of its return value; $\texttt{arg}_\texttt{i}$ are the names of its input arguments, with types $\texttt{Type}_\texttt{i}$; and $\texttt{stmt}_\texttt{i}$ are statements.

Functions must return a result by using **return statements**. As shown in the examples, simple functions without any conditional statements should end with a single return statement. Functions can use multiple return statements, e.g., in different branches of conditional (if/else or case) statements. A function that does not always return a value will produce a compiler error.

Functions can return only one value. To emulate returning multiple values, the function can return a struct (Section 7.2) instead. A function may have no arguments.

Functions can be parametric; see Section 10 for details.

Definition by assignment: Short functions consisting of a single expression can be defined more succinctly using the following syntax:

```
function RetType fname(Type1 arg1, ..., TypeN argN) = expr;
```

where `expr` is an expression of type `RetType` that computes the return value, and all other elements are as above.

Restrictions: Functions must obey two restrictions to be synthesizable to hardware:

1. No `Integer` arguments or an `Integer` result (`Integer`s may only be parameters, see Section 10).
2. No recursion (parametric functions can perform recursion on `Integer` parameters because all such recursive calls are unfolded at compile time, see Section 10).

```
function Bool and2(Bool a,
                   Bool b);
  return a && b;
endfunction

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

// Functions can have
// multiple return statements
function Bool maj(Bit#(3) x);
  Bit#(2) sum = {0, x[0]} +
                {0, x[1]} +
                {0, x[2]};
  if (sum >= 2) return True;
  else return False;
endfunction

// Shorthand syntax
function Bit#(1) p(Bit#(3) x)
  = x[0] ^ x[1] ^ x[2];
```

8.1 Combinational logic synthesis

Functions are always synthesizable into combinational circuits, i.e., circuits with digital inputs and outputs, where each output is a Boolean function of the inputs, and where 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. Combinational circuits are a basic building block of digital logic. Here, we discuss *why* Minispec functions are always synthesizable into combinational circuits, and sketch *how* this synthesis is done in practice.

Function properties: Minispec functions have three properties that differ from functions in other programming languages and that enable synthesis to combinational logic:

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 of the function (they may use global constants, however). Thus, given a particular set of input arguments, a function always produces the same output.
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. This is because *(i)* recursion is not allowed, and *(ii)* for loops have known and fixed iteration counts.
3. *Synthesizable arguments and output*: Each input and output of a function is always of a type that is representable using a fixed number of bits.

Why functions are synthesizable: Given the above properties, it is easy to show that all functions can be synthesized to combinational circuits. Since each input and output takes a fixed number of bits, we can always enumerate all input values, build a truth table (computing the output for each input), and implement the truth table as a combinational circuit (e.g., as a sum-of-products). While this is a simple proof, it is not *how* circuits are actually synthesized, as building a truth table and deriving an optimized implementation from it takes too much space and time for circuits with more than a few bits of inputs.

How functions are synthesized: 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 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. Specifically:

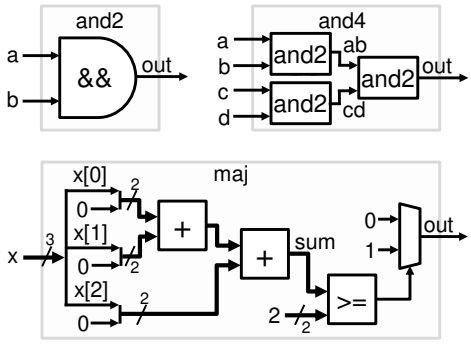

Figure 1: Synthesized functions from the examples in page 11.

- 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.
- All operators (Section 4) can be implemented using combinational circuits. Each operator is synthesized as a separate circuit.
- Conditional expressions translate to multiplexers: each ternary operator is a 2-to-1 multiplexer, and each case expression is an N-to-1 multiplexer.
- Concatenation expressions, struct creation expressions, and selection expressions that use statically known indices require wires but no logic: they are just combining or selecting bits of existing variables. Selection expressions that use dynamic indices require some logic (e.g., a barrel shifter).
- Functions are *inlined*: each function call instantiates a separate circuit implementing the function, which is then synthesized. (since there's no recursion, inlining always terminates).
- Variable assignments require no logic. Each assignment is simply naming the value (i.e., wires) of a particular expression, so it can be used somewhere else.
- If statements are translated to multiplexers: 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, and the multiplexer selects between the value in the if branch and the value in the else branch.
- Case statements are similar to if statements, requiring a multiplexer for each value assigned within the statement to select the right branch.
- For loops are *unrolled* (Section 5.3): Each iteration is expanded into a separate block, then synthesized as usual.

- A single return statement at the end of the function simply names the output of the function and requires no logic. In functions with multiple return statements, the output value is selected using multiplexers similar to if-else statements.

Synthesis with don't-care values: Don't-care values (represented by `?`, Section 2.7) denote values that are irrelevant. Don't-care values can be assigned to variables of any type. The compiler will pick a value for them. This flexibility often lets the compiler produce better circuits. For example, in the code `Bit#(4) x = ?; if (foo(y)) x = bar(z);`, the compiler can choose `?` to be `bar(z)` and eliminate the conditional altogether, optimizing the whole code snippet to `Bit#(4) x = bar(z);`.

9 Modules and Sequential Logic

Modules implement sequential logic, i.e., digital logic with *state*. Specifically, they implement single-clock synchronous sequential circuits, where 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). As shown in Figure 2, 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), register update their values, placing the FSM into the next state.

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 signals controlled by the enclosing module.

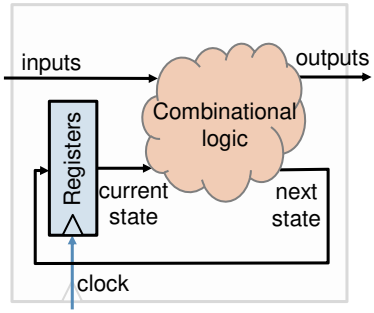

Figure 2: Finite State Machine.

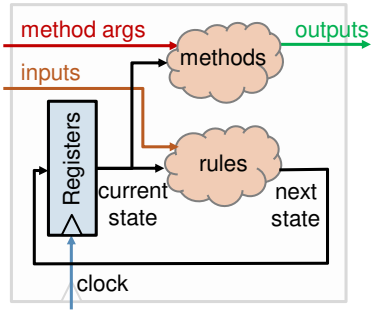

Figure 3: Basic module.

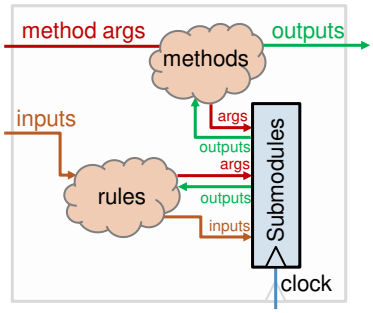

Figure 4: General module.

Figure 3 shows the structure of a basic module with registers but no other submodules. Comparing Figure 2 and Figure 3, the key difference 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. This separation *makes hierarchical composition easy* (Section 9.3). Figure 4 shows a general module that includes submodules other than registers. 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.

To see why rules and methods enable composition while avoiding combinational cycles, consider a different approach where we composed different FSMs directly as shown in Figure 5, 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 in Figure 5: 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.

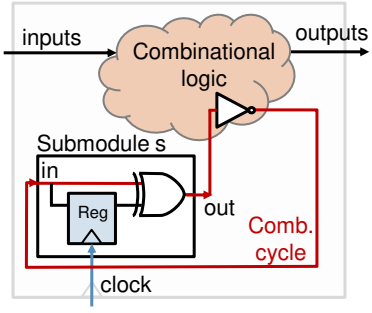

Figure 5: Example showing how composing FSMs by wiring their inputs and outputs can cause a combinational cycle.

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. It is thus 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.

Modules need not follow a strict hierarchy, but a hierarchical design has simpler semantics. Thus, we first focus on modules that follow a strict hierarchy (Section 9.3). We then describe the general semantics and additional mechanisms for modules that do not follow a strict hierarchy (Section 9.4).

9.1 Module syntax and elements

Modules are defined using the following syntax:

```
module ModType [(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*, $\texttt{arg}_i$, with types $\texttt{Type}_i$. These arguments can be constant values (e.g., used to set initial values), or other modules. The body of the module can declare submodules, methods, inputs, rules, and constants, whose syntax is explained below and shown in the examples to the right.

Modules can be parametric; see Section 10 for details.

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 $\texttt{arg}_i$ are the (optional) arguments to the module.

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; $\texttt{arg}_i$ are the names of its arguments, with types $\texttt{Type}_i$; and $\texttt{stmt}_i$ are statements.

Methods also support the same shorthand syntax as functions:

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

Use: A method may be called from the methods or rules of its enclosing module. The syntax for a method call is `submoduleName.methodName(arg1, ..., argN)` (Section 4.5).

```
// 8-bit counter that can be
// incremented every cycle
module Counter8;
  Reg#(Bit#(8)) count(0);

  method Bit#(8) getCount;
    return count;
  endmethod

  input Bool increment
    default = False;

  rule tick;
    if (increment)
      count <= count + 1;
  endrule
endmodule

// 16-bit counter built from
// two Counter8 submodules
module Counter16;
  Counter8 lo(0);
  Counter8 hi(0);

  method Bit#(16) getCount =
   {hi.getCount,lo.getCount};

  input Bool increment
    default = False;

  rule tick;
    if (increment) begin
      lo.increment = True;
      // Increment hi when lo
      // is about to overflow
      hi.increment =
       (lo.getCount == 255);
    end
  endrule
endmodule
```

Inputs specify external inputs that are controlled by the rule 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.

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.

9.2 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 circuitry (Section 9.5). 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.

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.

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.

9.3 Semantics of hierarchically nested modules

Suppose we impose two conditions on a design. First, modules follow a strict hierarchy, i.e., each module interacts only with the submodules it defines. 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 execute 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 the 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.

9.4 General semantics of modules (*advanced*)

Though the above semantics are simple, there are some cases for which strict hierarchical nesting can be too restrictive. For example, suppose we want to connect two modules through a FIFO (First-In-First-Out) queue: the producer module enqueues values into the queue, and the consumer module dequeues them. If we followed the hierarchical approach, the top-level module could instantiate three submodules: `producer`, `consumer`, and `queue`. Then, the rule in the top-level module could explicitly marshall outputs from `producer` into `queue` and from `queue` to `consumer`. But this can get cumbersome. Instead, we may want to have `producer` and `consumer` both directly interact with `queue`, as shown in the code to the right. But now `queue`'s methods and inputs are being used by two different modules, rather than by only its enclosing module. Moreover, there may be non-trivial interactions between them. For example, if the producer has a value ready but the queue is empty, we might want to let the consumer dequeue that value on the same cycle.

```
// Example: non-hierarchical
// module composition
module FIFO;
  input Maybe#(Bit#(4))
   enqueue default = Invalid;
  method Maybe#(Bit#(4))
   first = ...;
  method Bool isFull = ...;
  ...
endmodule

module Prod(FIFO outQ);
  ...
  rule produce;
    if (!outQ.isFull)
     outQ.enqueue = Valid(v);
  endrule
endmodule

module Cons(FIFO inQ);
  ...
  rule consume;
    let f = inQ.first;
    if (isValid(f)) ...
  endrule
endmodule

module TopLevel;
  FIFO queue;
  Prod producer(queue);
  Cons consumer(queue);
  ...
endmodule
```

General semantics: In general, rules in arbitrary modules can communicate in a single cycle, as long as doing so *does not introduce a combinational cycle*. The compiler will find a fixed order for the rules, and will emit an error on any cycle.

For instance, in our producer-consumer example, the FIFO queue module could have an `enqueue` input and a `first` method that returns that input if the queue has no buffered values. Because `consumer` calls `first`, `first` reads `enqueue`, and `producer` sets `enqueue`, `producer`'s rule is ordered before `consumer`'s. If there was another constraint in the system that required the reverse order (e.g., a queue going in the reverse direction), that would signal a combinational cycle, which would cause a compiler error.

Wires: In general, it may be desirable to split computation across rules and methods within the same module to avoid combinational cycles and code repetition, or to enforce a particular order. Wires are modules that allow rules and methods to communicate in the same cycle. They come in two flavors:

- `BypassWire#(T)` implements a wire that must be written on every cycle. It cannot be read until a rule writes to it.
- `DWire#(T)` implements a wire with a default value. If the wire is not written in a given cycle, reads to the wire return the default value. The wire has no memory: it "reverts" to the default value every cycle.

Wires have the same interface as registers: using the name of the wire yields its value, assignments with `<=` write to it, only rules may write to wires, and multiple writes are not allowed.

Wires introduce order constraints among rules and methods: the rule that writes to the wire is ordered before all rules and methods that read the wire.

9.5 Sequential logic synthesis

Sequential synthesis is a straightforward extension of combinational synthesis (Section 8.1).

Each sequential circuit has inputs and outputs as shown in Figure 4. Two of these inputs are *implicit* and not present in Minispec code: CLK, the *clock signal*, and RST, the *reset signal*.

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 Figure 6. Registers with no initial value (i.e., `RegU#(T)`) have no reset circuitry.

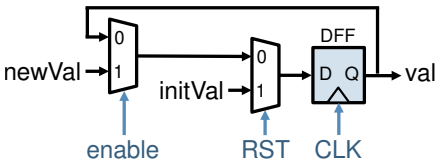

Figure 6: Example synthesized register with reset and enable circuits.

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 Figure 6.

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.

Finally, wires (Section 9.4) are synthesized exactly like inputs.

10 Parametrics 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, modules, structs, and type synonyms.

Parametrics enable writing *generic* hardware descriptions that are then concretized as needed. For example, we can write a single parametric function `add#(n)` that adds two `n`-bit numbers. Then, users of the function will call it with particular values of `n`, instantiating adder circuits of different widths. Without parametrics, each such adder would require writing a separate function.

In programming languages, parametrics (also known as generics, templates, or polymorphic types) primarily improve efficiency and safety: by specializing code at compile time, parametrics reduce work done at execution time and prevent type errors. In Minispec, parametrics play a more important role, because combinational logic has limitations that programming languages do not (it is not Turing-complete). Thus, *parametrics and compile-time specialization enable functionality that cannot be implemented with combinational logic*. For example, in a programming language without parametrics (like C), we could always write a procedure that adds two numbers of arbitrary bit-widths, e.g., by looping over their bits and adding them one by one. But we cannot implement a single function (combinational circuit) that adds numbers of arbitrary widths.

Parameters can be either `Integer`s or `type`s (including parametric types).

Parametric definitions follow a consistent syntax. Parameters are specified after the function name or module/struct/synonym type, enclosed in `#(...)`. Specifically:

- Functions: `function RetType fname#(PDef1, ..., PDefN) ...`
- Modules: `module ModType#(PDef1, ..., PDefN) ...`
- Structs: `typedef struct { ... } StructType#(PDef1, ..., PDefN);`
- Synonyms: `typedef Type NewType#(PDef1, ..., PDefN);`

and $\texttt{PDef}_\texttt{i}$ are the *parameter definitions*, each of which 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 (see discussion on *specialization* below).

```
// Doubles a Bit#(n) value
function Bit#(n+1) double
  #(Integer n)(Bit#(n) x)
  = {x, 1'b0};

// n-element FIFO of T's
module FIFO#(Integer n,
             type T);
 Vector#(n, RegU#(T)) elems;
 Reg#(Bit#(log2(n))) head(0);
 Reg#(Bit#(log2(n))) tail(0);
 method Maybe#(T) first;
 ...
endmodule
```

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 recursive on parameters. For example, function `rca#(n)` in Listing 1(a) calls `rca#(n-1)`. This recursion is unrolled at compile time, so it translates to a chain of function instances.

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

```
// n-bit ripple-carry adder with carry-in
// using a single parametric function
function Bit#(n+1) rca#(Integer n)
  (Bit#(n) a, Bit#(n) b, Bit#(1) cin);
 if (n == 1) begin
  let cout = (a & b) | (a & cin) | (b & cin);
  let sum = a ^ b ^ cin;
  return {cout, sum};
 end else begin
  let x = rca#(n-1)(a[n-2:0], b[n-2:0], cin);
  let u = rca#(1)(a[n-1], b[n-1], x[n-1]);
  return {u, x[n-2:0]};
 end
endfunction
```

(a)

```
// n-bit ripple-carry adder with carry-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 x = rca#(n-1)(a[n-2:0], b[n-2:0], cin);
 let u = rca#(1)(a[n-1], b[n-1], x[n-1]);
 return {u, x[n-2:0]};
endfunction
```

(b)

Listing 1: Two ripple-carry adder implementations using recursive parametric functions. (a) uses a single function, whereas (b) splits the base case into a separate specialized parametric function definition.

precedence over the general one. For example, an alternative definition of function `rca#(n)`, shown in Listing 1(b), includes a specialized definition of `rca#(1)` as a base case to stop recursion.

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.

Guarantees are important because we may want to use parametrics with expressions. For example, Listing 1(a) calls `rca#(n-1)`, and we must guarantee that all parameters are computed at compile time.

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 statements (if-else, case) 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.

To show how these rules combine, consider the example in Listing 1(a). Assume that some other code calls `rca#(3)`. This triggers the instantiation of `rca#(Integer n)` with `n=3`. Since `n == 1` is elaborated to `False`, the `if` branch is eliminated and all that remains is the `else` branch. This code calls `rca#(2)` (by elaborating `rca#(n-1)`) and `rca#(1)`. The compiler then instantiates `rca#(2)` (and leaves `rca#(1)` pending). `rca#(2)` is elaborated just like `rca#(3)`: the `if-else` is turned into the `else` branch, which calls `rca#(1)` twice. Finally, the compiler instantiates `rca#(1)`. `n == 1` is elaborated to `True`, so the `else` branch is eliminated and all that remains is the `if` branch. Since this branch does not call any functions, no more parametrics are needed and static elaboration terminates.

From the example above, note that recursion stops by relying on elimination of the `else` branch in `rca#(1)`. If this did not happen (i.e., if the `if`'s predicate could not be elaborated), then the `else` branch would trigger a call to `rca#(0)` and recursion would not end. (To avoid infinite recursion, the compiler limits recursion depth and emits an error when exceeded.)

11 Structure of a Minispec Design

A Minispec design consists of one or more *Minispec source files*. Each source file should be a plain text file with extension `.ms`. Each source file can define its own used-defined types (Section 7), functions (Section 8), modules (Section 9), and two other top-level statements not yet described: *constants* and *imports*.

Constants are declared just like variables and must be initialized at declaration. They may be read elsewhere but may not be

changed. They are not state (i.e., not like a global variable in other languages) and are simply a convenience to avoid hard-coding constant values throughout the design.

Import statements allow including the definitions from other files in the current one. Their syntax is `import file;`, where `file` is the name of the imported file without the `.ms` extension. For example, `import Example;` would import file `Example.ms`. `file` can be uppercase or lowercase. `file` cannot include a directory: the imported file must be in either the current directory or the compiler's search path (Section 13).

Import statements may appear anywhere (they need not be at the top of the file). An import statement is equivalent to including the contents of the imported file (and transitively, its own imported files) before the current file. A file can be imported multiple times, e.g., from several files in the same design. Import cycles (e.g., `file1.ms` with `import file2;` and `file2.ms` with `import file1;`). are not allowed, and will cause a compiler error.

Finally, Minispec code may also import BSV code through a **bsvimport** statement, with syntax `bsvimport BsvFile;`. Due to BSV conventions, `BsvFile` must be uppercase. This translates to an import statement in BSV, with its usual semantics.

12 System Functions

System functions are useful for simulation and debugging. They are used to display information, read and write data, and terminate the simulation. They are not synthesizable to hardware, and are only used when simulating a module.

All system functions begin with a dollar sign (`$`). System functions are called like normal functions. 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.

```
// Test harness for Counter8
// module from Section 9
import counter8;
module Counter8Test;
  Counter8 c;
  Reg#(Bit#(10)) cycle;
  rule tick;
    c.increment =
     (cycle[0] == 1'b1);
    $display("cycle ", cycle,
     ", count ", c.getCount);
    if (cycle == -1) $finish;
    cycle <= cycle + 1;
  endrule
endmodule
```

As shown in the examples 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 C's `printf`, using format strings with `%b` for binary, `%d` for decimal, and `%h` or `%x` for hexadecimal values.

```
$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"
```

Still, 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.

```
typedef struct { Bit#(8) red; Bit#(8) green; Bit#(8) blue; } Pixel;
Pixel cyan = Pixel{ red : 0, green : 255, blue : 255 };
$display(cyan);         // Prints "   65535"
$display(fshow(cyan));  // Prints "Pixel { red: 'h00, green: 'hff, blue: 'hff }"
```

`$display` terminates every printed string with a newline. If you do not want this behavior, use **`$write`**, which uses the same syntax as `$display`.

Other system functions: Minispec can use other system functions from BSV, e.g., to read standard input or to read and write data from/to files. BSV system functions are described in Section 12.8 of the BSV reference. They are rarely needed.

13 Minispec Tools

Minispec includes an easy-to-use toolset to simulate and synthesize circuits. We first describe its command-line interface, than its Jupyter interface.

13.1 Command-line tools

`ms` is Minispec's high-level command-line interface. `ms` provides four commands:

- `ms eval [<file>] <expression>` — Evaluate expression
- `ms sim <file> <module>` — Simulate module
- `ms synth <file> <function/module> <synthArgs>` — Synthesize function or module into gates
- `ms help` — Print help message

The `<file>` argument should be a Minispec source file with the target function or module. The file argument is optional for `ms eval`, as it is not needed if the expression does not call any user-defined functions. Arguments should be quoted as needed to avoid being interpreted by the shell.

For more details, run `ms help`.

Internally, `ms` uses two lower-level tools: **`msc`**, the Minispec compiler, which compiles functions or modules into BSV, Verilog, or a simulation executable; and **`synth`**, a synthesis tool for Minispec and Bluespec circuits that leverages the yosys open-source synthesis suite and the FreePDK45 open-source 45nm standard cell library to produce optimized gate-level circuit implementations, and to analyze their delay and area. These tools can be used directly; run `msc -h` or `synth -h` for usage instructions.

```
# Evaluate "2 + 3"
ms eval "2 + 3"
# Evaluate add#(4)(2, 3), where
# add is defined in file add.ms
ms eval add.ms "add#(4)(2, 3)"

# Simulate module TestCounter
# in file counter.ms
ms sim counter.ms TestCounter

# Synthesize function add#(4)
ms synth add.ms "add#(4)"
# Synthesize module Counter
ms synth counter.ms Counter
```

13.2 Jupyter interface

Minispec is integrated with Jupyter notebooks. Minispec Jupyter notebooks follow the usual conventions for notebooks in other languages: Minispec code can be spread across multiple *code cells*, which the notebook user can execute individually or in sequence. A code cell can use or redefine functions, modules, types, or constants defined in previously executed cells. Redefined functions or modules can be used by later code, but previously executed cells still use the old definitions. In a Minispec notebook, "executing" a code cell only checks the code for errors. To provide additional capabilities, cells can include additional built-in commands, called *magics* in Jupyter terminology.

Magics: Minispec notebooks provide four magics: `%%eval`, `%%sim`, `%%synth`, and `%%help`. Each of these magics has the same purpose as the `ms` command with the same name (Section 13.1), and follows a similar syntax.

There are two syntax differences between Jupyter magics and `ms` commands. First, Jupyter magics do not take a file argument; instead, they work on the code from executed cells. Second, Jupyter magics do not need quotes on expressions or parametric functions or modules, as they are not interpreted by the shell.

The Jupyter notebook tutorials on combinational and sequential logic serve as detailed examples on using Jupyter notebooks and the different magics.

14 Copyright, License, and Contributing

The source code for this reference is publicly available in the Minispec repository, along with the Minispec compiler and toolset. If you find mistakes in this document or want to contribute improvements, pull requests are welcome!

This document was produced on 2019-10-10 from source version master:64:78e7e5c:clean .