Sandbox Language Reference Manual

MEGAN FILLION (mlf2179)
GABRIEL GUZMAN (grg2117)
DIMITRI LEGGAS (ddl2133)

October 16, 2017

Preface

For the reference manual of sandbox, a hardware description language for writing circuits in terms of nested blocks, see below.

1 Lexical Elements

Identifiers

An identifier is a letter followed by any union of ASCII specified characters and digits:

- Letters: `[a-z]*`
- Digits: `[0-9]*`

Examples include:

- x
- x10ten
- x1023

Keywords

The following words are reserved for language-specific use: `int`, `sandbox`.
Note: The absence of keywords like `if` and `for` is explained in Section 4.

Comments

The comments follow the style of Java. We have:

- Single line comments begun by "`//`"
- Multiple line comments demarcated by "`/*`" and "`*/`"

2 Data Types

int(k)

In the style proposed by EHDL, we use `int(k)` for $k > 0$ to denote a k-bit bus.

Arrays

Again, we look to Java for array definitions. For example, `int(2)[4] x` is an array of 4 2-bit values.

Tuples

We allow for tuples of mixed types. For example, `{int(1)x,int(2)y,int(4)z}` gives a tuple containing a 1-, 2-, and 4-bit integer. The use of tuples for composing functions will be elaborated on in Section 4.

3 Expressions and Operations

We support the standard logic gates:

- And: `&&`
- Or: `||`
- Exclusive OR: `^`
- Not: `!`

We also support comparators: `<`, `>`, `==`, `<=`, `>=`
The "dot" operation will allow the programmer to refer to inputs and outputs of a function; this will be elaborated in Section 4.

The most important operation in `sandbox` is the "map" operation denoted by `->`. Map is used to compose functions. This operation is unique to `sandbox`. Consider the following example in which one tuple is mapped to another:

```
{a1, a2, a3} -> {b1, b2, b3}
```

Here, the contents of a1 will be copied into the variable b1, a2 into b2, and so on. This form permits the user to condense copious amounts of code into a few lines. We will use, as seen in Section 4, that maps can be used to build increasingly complex combinational circuits. The left and right tuples must have consistent types.

4 Functions

Functions act as code blocks, meaning they map a list of inputs to a list of outputs. The methods created by the programmer in sandbox will follow some conventions of Java, namely that of demarcating the function by braces, separating statements by semicolons, etc.:

```
(type out1,...,type outN) nameOfMethod (type in1,...,type inN){
        stmt1;
        stmt2;
}
```

As seen above, the variables in the parentheses preceding the method name would represent the output variables and the ones in the succeeding parentheses would be the input arguments. Functions in sandbox do not have a return statement; it only matters if every specified output is defined by the end of the execution of the function.

The following is a fulladder method. It demonstrates how the map operation ”maps” inputs to outputs”

```
(int(1) sum, int(1) carry) fulladder (int(1) a, int(1) b, int(1) cin){
        a ^ b ^ cin -> sum;
        (a && b) ^ (cin && (a ^ b)) -> carry;
}
```

Whereas we could have used a simple “=” to set the values of the outputs, this approach is not conducive to defining more complicated functions in terms of one another. For example, consider trying to write a function for a 4-bit ripple adder:

```
(int(4) s, int(1) carry) add4 (int(4) a, int(4) b, int(1) cin){
        FA0, FA1, FA2, FA3 = fulladder;

        {a(0), b(0), cin}          -> {FA0.a, FA0.b, FA0.cin};
        {a(1), b(1), FA0.carry}  -> {FA1.a, FA1.b, FA1.cin};
        {a(2), b(2), FA1.carry}  -> {FA2.a, FA2.b, FA2.cin};
        {a(3), b(3)], FA2.carry} -> {FA3.a, FA3.b, FA3.cin};
        {FA0.sum, FA1.sum, FA2.sum, FA3.sum, FA3.carry}
                -> {s(0), s(1), s(2), s(3), carry};
}
```

First, sandbox allows for the programmer to define multiple “instances” of a function in order to help them conceptualize separate code blocks and how they relate, but at compile time these instances will be replaced with the single function definition. The next four lines represent the use of each fulladder in a 4-bit ripple adder. The zeroth bits and the carry-in are mapped to the inputs of the zeroth full adder. Then its carry and the next two input bits are mapped to the inputs of the first fulladder. This continues for all four fulladders. Then, all the outputs of the fulladders are mapped to the outputs of the function itself. At compile time, this visually appealing representation of the circuit relations will be rewritten as function compositions.

4.1 sandbox

All compilable programs need a `sandbox` method which is the "main" method. Its inputs and outputs define the inputs and outputs of the entire circuit.

4.2 Library Functions

We plan to include a library of builtin functions corresponding to commonly used circuits. Some will include:

- `half_adder`
- `full_adder`
- `mux`
- `encoder`

We also plan to include builtin code blocks to assist with control flow: `if/else`, `for`, `switch`, `case`.

5 Program Structure and Scope

Programmers write functions corresponding to blocks of circuitry which can be nested to form increasingly complex circuits. Scope is delimited by `{}`, `[]`, `()` as in Java. Every program must have a `sandbox` method, which is the "main." Any method written above it can be used in the "sandbox" environment.

6 Sample Program

We provide a sample program which incorporates the methods defined above.

```
(int(1) sum, int(1) carry) fulladder (int(1) a, int(1) b, int(1) cin){
        a ^ b ^ cin -> sum;
        (a && b) ^ (cin && (a ^ b)) -> carry;
}

(int(4) s, int(1) carry) 4add (int(4) a, int(4) b, int(1) cin){
        FA0, FA1, FA2, FA3 = fulladder;

        {a(0), b(0), cin}         -> {FA0.a, FA0.b, FA0.cin};
        {a(1), b(1), FA0.carry}   -> {FA1.a, FA1.b, FA1.cin};
        {a(2), b(2), FA1.carry}   -> {FA2.a, FA2.b, FA2.cin};
        {a(3), b(3)], FA2.carry} -> {FA3.a, FA3.b, FA3.cin};
        {FA0.sum, FA1.sum, FA2.sum, FA3.sum, FA3.carry}
                -> {s(0), s(1), s(2), s(3), carry};
}

// sandbox is the main function
(int(4) s, int(1) carry) sandbox (int(4) x, int(4) y, int(4) z){
        ADDER1, ADDER2 = 4ADD;
        {x, y, 0} -> {ADDER1.a, ADDER1.b, ADDER1.cin};
        {ADDER1.s, z, ADDER1.carry} -> {ADDER2.a, ADDER2.b, ADDER2.cin};
        {ADDER2.s, ADDER2.carry} -> {s, carry};
}
```

References

[1] Paolo Mantovani, Mashooq Muhaimen, Neil Deshpande, Kaushik Kaul, EHDL Language Reference Manual, http://www.cs.columbia.edu/~sedwards/classes/2011/w4115-fall/lrms/EHDL.pdf (2011).

[2] Trevis Rothwell et al., The GNU C Reference Manual, https://www.gnu.org/software/gnu-c-manual/gnu-c-manual.html.

[3] VHDL Reference Manual, http://www.ics.uci.edu/~jmoorkan/vhdlref/Synario\%20VHDL\%20Manual.pdf (1997).