FindTune Compiler Specification/Guide

Ilhan Bok, Ben Holzem, Tristan Wentworth

ECE 554, Spring 2020

Last updated: 4/3/20

V3.0

1.0 Overview

This guide attempts to be as direct, short, and helpful as possible, but may not be comprehensive. Be sure to message #compiler on Slack if any clarification is needed. **Please report any bugs found to ensure they are fixed in upcoming releases.** The output of the program should be included in any bug reports.

The best way to learn the language is to skim this spec, then load the compiler and try experimenting, referring back here or contacting the compiler subgroup when necessary. Remember to read the output of the compiler for direct and hopefully useful information.

1.0.1 Usage

To use the compiler, download `compiler.py`, `ft_scanner.py`, `ft_parser.py`, and `ft_generator.py` then create a new file (e.g. `high_level.ft`), write FindTune code in `high_level.ft`, then run

Then run `python3 compiler.py -p <prefix> high_level.ft`

Note that Python >=3.0 is required, for maintainability reasons.

The optional prefix creates a directory named after the prefix where intermediate files (scanner and parser output, first pass generation with assembly comments). **Please report any bugs found to ensure they are fixed in upcoming releases.**

1.0.2 Planned Features

1. Suggestions welcome

1.0.3 Implemented Features

1. Parentheses for grouping math and booleans
2. And, or, and not
3. Removed modulo (%) operator (nested for loops do the same thing)
4. Remove boolean literals (i.e. true, false)
5. Compare boolean expressions (not just math equality)

2.0 Language Syntax

FindTune supports the following, which are detailed below

1. *Comments (both single and multiline)*
2. *And, or, not*
3. *Integers*
4. *Bit strings*
5. *Equality*
6. *FPGA directives*
7. *If statements*
8. *If else statements*
9. *For loops*

Parentheses must be included around equality statements, or to group subexpressions. **Everything else is evaluated from left to right**.

```
((x == 1) || (x == 2)) && (x == 3) # valid
(!(y == 1)) # valid
((((z == 1) == (x == 1)))) # valid
(z == x) # invalid (allowed in parser only)
(1 == x) # invalid (allowed in parser only)
```

2.0.1 Comments

```
# this is a single line comment
```

Single line comments extend until the end of the line, from where they start

```
:(
this is a multiline comment. It is sandwiched between
the frowny and smiley. You do not need to close
multiline comments, but they will extend until the end
of the file if you do not
:)
```

2.0.2 And, Or, Not

```
(x == 1) && (y == 2) # valid
(x == 3) || (y == 3) # valid
!(i == 1) # valid
(!i == 1) # valid, same as !(i == 1)
((x == 2)) # valid
```

Remember that all operators **(including !) parse from left to right**, unless parentheses are included

2.0.3 Integers

Integers can be used to perform equality comparisons. Integers cannot be negative, for now. Leading zeros are removed.

```
0 # valid
0000 # valid
0001 # valid
12394 # valid
```

2.0.4 Bit Strings

Bit strings are surrounded by angled brackets. They may contain only zero or one.

```
<00101> # valid
<12345> # invalid
<00000000> # valid
001001 # just an int, not a bit string
```

2.0.5 Equality

Compare if two bools or two ints are equal, but there is no mixing.

```
val == 2 # valid
(x == 1) == (x == 2) # valid
(num == 5) == 5 # invalid
```

2.0.6 FPGA Directives

FPGA directives induce an action in the hardware.

For now, only setOctaves is implemented in the generator, but others can be added per the ISA if needed.

```
fpga.enableSample # valid
fpga.setOctaves(<00001110>) # valid
fpga.wait(123) # valid
```

Actions may include:

1. enableSample
2. disableSample
3. displayState
4. enableGraphics
5. disableGraphics
6. freeze
7. unfreeze
8. enableAutotune
9. disableAutotune
10. setOctaves(<bitstring>)
    a. <bitstring> must contain one bit (1 = on, 0 = off) for each octave being output, from lowest to highest
11. wait(math)
    a. Wait must contain either an int or bitstring

2.0.7 If statements

```
if (boolean) {
  // an if statement, if else statement, for loop, or
  // directive
}
```

Note that the braces are required.

2.0.8 If else statements

```
if (boolean) {
  // an if statement, if else statement, for loop, or
  // directive
} else {
  // an if statement, if else statement, for loop, or
  // directive
}
```

Note that the braces are required

2.0.9 For loops

```
for [variable] in ([math], [math]) {
  // an if statement, if else statement, for loop, or
  // directive
}
```

Where [variable] can be any variable name not starting with a number (but can contain or start with an underscore), and math is either an int or bitstring. The scope of the variable is within the loop itself.

Note that the braces are required