# SimpleGV Input Set Format

"Simple" is no longer the best adjective to describe this format. "Alpha" is probably more appropriate. Our plan is to add features to this format through the beta-testing of the circuit tester, then finalize the syntax and feature set after we get feedback from the users. In addition, as we learn more about the JavaCC "context" feature, we plan to relax the rules a little and do things like remove the concept of "keywords". Unless we find a big problem, valid " `SimpleGVFormat` " files will be valid "Regular" `GVFormat` files.



## Basic Syntax

The basic syntax of the `SimpleGV` format is demonstrated by the following example :

```
#> InputSetLoader: gvFormat.InputSetLoaderSimpleGVFormat

# This file tests a basic half adder with one-bit inputs named "A" and
# "B", and one-bit outputs named "Sum" and "Carry"

BEGIN test1
  INPUTS
    InputA 1
    InputB 1
  OUTPUTS
    Sum 0
    Carry 1

BEGIN test2
    INPUTS
      A [0, 1]
      B [0, 1]
    OUTPUTS
      Sum [0, 1, 1, 0 ]
      Carry [0, 0, 0, 1 ]
```

Notes:

- The "magic line" beginning with "`#> InputSetLoader:`" identifies which `InputSetLoader` (and, therefore, which input format) to use.
- Everything on a line after a `#` is ignored.
- Keywords are currently ALL CAPS to decrease the likelihood of circuits using a keyword as a pin name.
- As shown in `test2`, the user can specify a list of values for an input pin. This results in the definition of multiple tests (four, in this case). Notice that each output then list contains one value for each test. The multiple tests are ordered as if they were defined by a series of nested loops with the first input listed in the

outermost loop, and the last in the innermost loop. (If you get confused, run the circuit tester in `--verbose` mode to see how the tests get ordered.)
- At present, only lists can be broken by a newline. All other expressions must be on one line only. (This avoids the use of "END" tags.)

# Intermediate Syntax

This example demonstrates some intermediate features, such as

- named value lists,
- extended integers (e.g., `2^15+1`),
- instructions to ignore certain output pins (i.e., `CarryOut`),
- negative numbers, and
- use of a Java class to compute output values. (See `jlsCircuitTester` documentation for instructions on how to write custom `OutputSet`s.)

```
#> InputSetLoader: gvFormat.InputSetLoaderSimpleGVFormat

# Name of java class that computes the correct answer
OUTPUT_SET_TYPE SHARED UnsignedAdderOutputSet

NAMED_VALUE_LISTS
  # The sum of any two smallPositive integers (or sum of any two
  # plus a carryIn of 1) will not cause an overflow.
  smallPositive [ 0, 1, 2, 3, 4, 5, 10, 15, 16, 17, 30000, 2^15 - 1 ]
  allPositive   [ smallPositive, 2^15, 2^15 + 1, 2^15 + 16385,
                  2^16 - 2, 2^16 -1 ]

  # This circuit doesn't use negative numbers.  They are included here
  # only to serve as an example.
  lowNegative SIGNED(16) [ -1, -2, -3, -5, -10, -15, -16, -17, -14512,
                           -2^14 + 1 ]
  allNegative SIGNED(16) [ lowNegative, -2^14, -2^14 - 1, -2^14 - 8192,
                           -2^15 + 2, -2^15 + 1, -2^15 ]


# This first tests uses only inputs that won't cause overflow.
# The CarryOut pin should always be 0; however, by ignoring it
# I can completely test the addition component of the circuit
# regardless of whether overflow works.
BEGIN no_overflow
  INPUTS
    InputA smallPositive
    InputB smallPositive
    CarryIn [ 0, 1 ]
  OUTPUTS
    # Tell JLSCircuitTester to ignore value of CarryOut output pin
    CarryOut EXCLUDE

# "Requiring" that CarryOut be explicit set just helps detect bugs
# in the code for UnsignedAdderOutputSet.
BEGIN overflow
  INPUTS
```

```
    InputA allPositive
    InputB allPositive
    CarryIn [0, 1]
  OUTPUTS
    CarryOut REQUIRE
```

Notice that when using negative numbers for input (and for output), you must explicitly specify the width. This specification is necessary so the program knows where to put the sign bit.

# Memory Input/Output

Users can specify the initial or expected contents of memory either (1) inline in the test file, or (2) in a separate file.

## Inline

Data specified in-line begins with the `DATA` keyword. It is specified in one of two ways: The first is a list of pairs of address and data values: For example:

```
MEMORY example1 DATA [0 100, 1 122, 2 984, 100 734, 200 212]
```

Note that (1) the addresses need not be consecutive, and (2) there is no comma between the address and the data.

The second specification method is a short-cut in which an address is followed by a list of values for the following consecutive addresses. For example,

```
MEMORY example2 DATA [0 [0, 100, 200, 300, 400]]
```

is equivalent to

```
MEMORY example2 DATA [0 0, 1 100, 2 200, 3 300, 4 400]
```

And, of course, the two styles can be mixed:

```
MEMORY example3 DATA [0 [0, 100, 200, 300, 400], 10 109, 12 434, 15 189, 20 [101, 102, 103, 104, 105], 43 55]
```

## File

A memory description can also be placed in a separate file.

```
MEMORY example4 FILE "memory_v1"
```

There are currently two file formats for memory, `JLS` and `SimpleGV`, although the user can easily add more. The `JLS` input format is format used by `JLS` 's batch mode. It is simply a sequence of lines containing an address and a value, both in hex. Here is one example. Notice that

- blank lines and comments are allowed,
- addresses need to be consecutive, and
- *all* numbers are in hex.

The `SimpleGV` format is identical to the format for inline data. (See this example.)

The input format of a file is determined as follows:

1. If the first non-blank line in the file begins with "`#> MemoryFileLoader:`" or "`#> MFL:`", then the next word on that line is assumed to be the Java class that parses the file. For example, a file that begins with "`#> MFL: edu.gvsu.cis.kurmasz.jls.test.gvFormat.SimpleGVMemoryFileLoader`" specifies the `SimpleGV` input format.
2. If the file itself does not specify the loader, then the parser looks between the `FILE` keyword and the file name for the loader to use: For example, `MEMORY example5 edu.gvsu.cis.kurmasz.jls.test.gvFormat.SimpleGVMemoryFileLoader "memoryInput_sgv"`
   specifies that the `SimpleGV` loader should be used, unless the file begins with a "magic" "`#> MFL:`" line specifying a different loader.
3. Finally, the `JLS` format is used by default.

See the `jlsCircuitTester` documentation for details on how to write a new memory file loader.

# Programmed Value Lists

Programmed value lists are Java methods the produce a list of numbers according to some algorithm. The `SimpleGV` format currently supports 6 programmed value lists. Below is a brief description of each. Each name is also a link to the underling class's javadoc, which provides details on the algorithm and its parameters.

`RANGE(int begin, int end, int step=1)`
: Generates a list of numbers in the range `[begin, end]`. The values generated correspond to the values taken on by `i` in the loop `for(i = begin; i <=end; i+=step)`.

`CORNERS(int begin, int end, int step=1)`
: Generates a list of numbers in the range `[begin, end]` that are of the form $2^i-1$, $2^i$, or $2^i+1$. For example, `CORNERS(8, 31, 1)` would produce `{8, 9, 15, 16, 17, 31}`.

`ECORNERS(int begin, int end, int step=1)`
: Generates a list of numbers in the range $[2^{begin}, 2^{end})$ that are of the form $2^i-1$, $2^i$, or $2^i+1$. For example, `ECORNERS(3, 5, 1)` would produce `{8, 9, 15, 16, 17, 31}`.

`WILDCARD(int fixed, int wildcards)`
: Generates a list of numbers based on "wildcards". For example, the binary value the binary value `10**` is expanded into the set `{1000, 1001, 1010, 1011}` (which is returned as the array `{8, 9, 10, 11}`). Because we can't represent `1`s, `0`s, and `*`s with a single binary number, `WILDCARD` requires two parameters. See the `Wildcard` javadoc for details.

`RANDOM(int min, int max, int amount)`
: Generate a list of `amount` random integers containing values between `min` and `max` (inclusive).

`UNIQUERANDOM(int min, int max, int amount)`
: Generate a list of `amount` *unique* random integers containing values between `min` and `max` (inclusive).

# Common Problems

- The `Output Set` is responsible for specifying which output pins, registers, and/or memories are to be checked. It does not assume that every register, output pin, and memory in the circuit is to be checked. Thus, if the user forgets to list a pin when writing a test file, the test will always pass. The same thing will happen, if the programmer of a custom `OutputSet` forgets to add a value for an output to be tested.

# Grammar

The formal grammar for the `SimpleGV` format (generated by `JJDoc`) is here.

Keywords:

- BEGIN
- GATE_DELAY
- ELEMENT_DELAY
- SIGNED
- UNSIGNED
- FIXED
- INPUTS
- OUTPUT_SET_TYPE
- SHARED
- MEMORY
- FILE
- DATA
- NAMED_VALUE_LISTS
- OUTPUTS
- INCLUDE
- EXCLUDE
- REQUIRE
- RANGE
- ECORNERS
- CORNERS
- UNIQUERAND
- WILDCARD
- RANDOM

---

Last modified: Wed May 21 11:41:30 EDT 2007