Appendix B

Abbreviated User's Guide.

1 Overview

This document introduces two tools, CIRC and GEN.

The first tool, CIRC reads an input netlist and performs analysis upon it, outputting either statistical information, or acting as a filter to convert the netlist to an alternative format.

The second tool, GEN, takes a parameterization of a circuit as a program written in the SYMPLE language and creates a netlist which corresponds to the parameterization program.

Though GEN and CIRC are separate tools, their usage is highly related. Many of the most useful products of the research from which they arise is the interaction between the characterization of a circuit and the subsequent generation of a similarly parameterized circuit. Thus, it is more appropriate to document their usage in a single document.

This user's guide is organized as follows. Section 2 discusses the "characteristics" of a circuit. These characteristics then form the basis for the output of CIRC and for the input to GEN. Section 3 describes how to use CIRC to analyze or filter a circuit. Section 4 describes basic usage of GEN to create combinational and sequential circuits. Section 5 discusses more advanced usage of GEN such as the problems involved in modifying existing scripts or scripts from CIRC.

2 Circuit Characteristics

This section defines the terms which will be used throughout the document to describe characteristics and parameters of circuits.

The most basic parameters of a circuit are the following:

name The filename in which the netlist is stored. CIRC will look for name.blif, name.blf, or $MCNCDIR/k/name.blif.

k The lut-size (maximum fanin) of the design.

size The size of a circuit. The size of a circuit is defined as the number of "countable functional nodes" in a graph-theoretic sense, hence it is the sum of the number of (see below) PIs (primary inputs), LUTs (or logic nodes), and DFFs (flip-flops). Primary outputs are not counted, because we consider this to be an attribute rather than a separately named node.

nPI The number of primary inputs designated in the .inputs line of the input or output netlist.

nPO The number of primary inputs designated in the .inputs line of the input or output netlist.

nDFF The number of D-type flip-flops which are defined in the input or output netlist. Currently the only type of sequential logic element which is understood by CIRC and GEN is the DFF with no defined preset or clear.

nEdges The number of edges in the circuit-graph. Equivalently either the sum over all nodes x of fanin(x), or the sum over all nodes of fanout(x).

Currently the tools recognize only a *single* clock. In the case of CIRC this means that all clock-inputs are ignored, and replaced by a single primary-input called "clock," effectively forcing all DFF to use the same clock regardless of the design specification. Similarly, GEN will output circuits of the same form.

nCC The number of connected components: essentially the number of completely separate circuits which are defined in the same file. This value is output by CIRC but GEN will only output circuits which are fully connected (one component).

unusable nodes The number of nodes which do not affect any PO in an input design. These are deleted by CIRC before processing, and should not every be produced by GEN.

unreachable nodes The number of nodes which (recursively) cannot be reached from a PI, and hence will never have a logical value. These are also deleted by CIRC before processing, and should not be produced by GEN.

The basic element in CIRC and GEN processing is the combinational circuit, using the combinationl delay of the circuit as an important point of reference. Thus we have several items defined on the basis of combinational delay.

delay Combinational delay is defined, for all nodes x in a circuit, as follows: delay(x) = 0 if x is either a PI or a DFF. delay(x) = 1 + MAX{delay(y)}, for all fanins y to x; essentially a standard unit-delay model of combinational delay. The delay of a circuit is then the maximum combinational delay over all nodes x in the circuit.

shape The combinational shape of a circuit is defined as the distribution of node combinational delays. It is vector of length delay + 1 (0..delay). In a purely combinational circuit, shape[0] is necessarily nPI, and shape[delay] is no more than nPO (though it need not be nPO, because nodes of earlier delay can be designated as POs). Thus a shape of [4, 8, 3, 2] specifies a circuit with 4 PIs, 8 nodes which have inputs only from the PIs, 3 which have at least fanin from delay level 1, and 2 which have at least one fanin from level 3.

nBot The number of "bottom" nodes in the shape distribution. Though nBot is redundant information in general, it is referred to in various places, and is used as an intermediate calculation in the creation of a default/random shape vector by GEN.

POshape In the same way that shape[] is defined, we can have a vector to represent the distribution of POs in the circuit. This is both reported by CIRC and used as a parameter by GEN.

edges Also given the combinational delay of each node in the circuit-graph, we can define a distribution based on the edges of the graph. The length of an edge (x, y) is defined as delay(y) - delay(x), yielding a vector [0..delay] with sum the number of edges in the graph.

Fanout is also both an important characteristic of a circuit and parameter to generation. We have

max-fanout The maximum fanout (number of edges leaving) any node x in the circit-graph.

outs A vector representing the distribution of fanouts in the circuit. The vector is of length [0..max-fanout], is non-zero in the last element (or max-fanout is incorrect), and outs[0] $\leq$ nPO necessarily.

We also have a number of statistics which are output by CIRC which are calculated from the above metrics. For example, the average fanin/fanout, and the average fanin/fanout of each combinational delay level and associated standard deviations. These are not documented further at this time. However, they appear in the GEN defaults files as intermediate calculations when creating a default out-degree distribution.

Throughout CIRC and GEN, sequential circuits are described as a collection of combinational circuits. Within CIRC, a circuit is processed into *sequential levels* and we define "ghost" edges which cross the boundary between one combinational sub-circuit (sequential level) and another.

level The sequential level of a node x in a sequential circuit is defined as the *minimum* number of DFF on a directed path from any primary input. More formally, level(x) = 0 if x is a PI, level(x) = 1 + level(fanin d) if x is a DFF, and MIN(level(y), over all fanins y to x) otherwise.

back-edge An edge in the circuit which connects x to a node y at a *preceeding, different* sequential level. In other words, a feedback edge.

bottom-node A node which has all fanout-edges as back-edges is at the "bottom" of its combinational sub-circuit. The number of such nodes is relevant in the understanding of sequential circuits and how to generate them.

invisible-node Sometimes, especially when building a clock splitter or similar structures, it is possible to have a set of registers and logic which is self-contained and feed purely from itself (no PIs affect the output) and just outputs values. This is different from being unreachable (see above), because the value *is* affected by the clock. These nodes are not deleted by CIRC, becasuse they are important to the understanding of circuits, but they have to be treated as special cases to our basic model of a circuit because they have no real concept of sequential level.

forward-edge A forward edge is one which follows the normal rules of combinational delay when level is ignored, or which connects to a DFF at the next sequential level. That is, an edge which is not a back edge as previously defined.

This allows us the concept of level-shape and a distribution of back edges between levels (i.e. difference in sequential levels), but this will not be discussed at this time.

Because sequential circuits are generated at the base level as combinational circuits, we need a mechanism to define future back edges and forward edges to a DFF. This is done in terms of ghost input and output edges:

GI, nGI Each node in a hierarchically defined circuit or sequential input design will have its fanin divided into nodes which appear in the same sub-circuit and those which do not, called *ghost inputs (GI)*. The number of ghost inputs to a node (nGI) is defined for each node, and the total number of ghost inputs over all nodes is nGI for the circuit. $nGI(x)$ is always strictly less than kin, as one input to each node must be "real" for it to belong to one sub-circuit.

GO, nGO Similarly, we have ghost outputs, and nGO.

GIshape In the same way that shape[] is defined above, we have the concept of a distribution vector of GIs. Note that, when talking about sequential sub-circuits, we count nDFF in the shape profile, not in the GI shape profile, mainly due to internal details of shape generation beyond the scope of this document.

GOshape Similarly, we can store the combinational delay of each ghost output edge. as the delay of its source. Note, though it is required that any ghost edge has either dst.type == DFF or delay(src) < delay(dst), it is not necessarily true that delay(src) == delay(dst) - 1, because of the MAX relationship in the definition of delay.

Note that for a final circuit nGO(C) == nDFF + nGI(C) necessarily, as each ghost output corresponds to exactly one ghost input, or else eventually feeds one DFF.

It is important to note that PI and PO refer to nodes, whereas GI and GO refer to ports in or out of nodes, more like edges in a graph.

One final characteristic of circuits is the reconvergence number, or **rnum**. This is output by CIRC, but is not used by GEN so will not be discussed further here. Details on reconvergence calculation are contained in the published papers.

3 Using CIRC.

CIRC is a command-line based tool. The calling sequence is as follows:

```
circ in=<name> [k=<kin>] [options] [xnf | verilog | tdf | adl | gen] [out=<file>]
```

The only required parameter is the name of the file to be analyzed. The input format to CIRC is exclusively blif, so all files must be externally converted to blif before processing. CIRC will search in the current directory for the files name, name.blif, or name.blf, then search in the MCNCDIR (environment variable) directory in the 'k' subdirectory (k defaults to 4 if not specified).

The output of circ is to stdout. This can be overridden with the out= option. Note that the xnf,verilog,tdf,gen options automatically set out to 'name' with the appropriate file extension.

3.1 Using CIRC for format conversion.

To use CIRC as a filter to convert test.blif to either xnf, verilog or ahdl (tdf) formats, use the following syntax:

```
circ in=test <format>
```

where format is one of {xnf, xnfROM, verilog, tdf, ahdl} (tdf and ahdl are the same thing).

Note that k will automatically be set to 4, because all formats are output using the 4-LUT primitive. The program will fail if any node exists in test.blif which has fanin>4.

Currently, the ahdl and verilog formats output only NAND gates for LUTs. The xnf option will output ROM-based output if the option is specified as "xnfROM," but input which originated from GEN will still have only NAND gates defined (i.e. will simply be ROMs which define a NAND gate).

3.2 Using CIRC for statistical output.

Currently the "dump" format is the most stable form of output. There are other options available, but they are obsolete. The command:

```
circ in=test dump
```

will output a complete description of the design test. The output format is such that it is easy to use AWK, SED or GREP to extract and build tabular information from the output files of multiple circuits.

We will go through the output for an MCNC circuit, bbrtas:

First, there is some informational output to stderr (which does not appear in the output file). This gives the version and compile date of the software, and warning/error conditions encountered. In this case, bbrtas has a single unusable node "pclock." This is not a problem, CIRC is just noting that it dropped pclock as an unusable input when it replaced all clocks by the global signal 'clock.'

```
CIRC 2.2, compiled Fri May 24 12:04:30 PDT 1996.
Analysis of bbrtas beginning at Mon May 27 14:07:13 1996
Warning:  Deleting PI pclock because it does not drive a primary output
Warning:  (For further such nodes, use verbose option)
Warning:  Circuit has 1 unusable nodes
```

Note the mention of a command-line option "verbose" to see more detailed information, especially about error and warning messages.

Within the output file, we begin with introductory output, listing the options and the actual file name used. The filename is important because we used an MCNC circuit; this shows that we picked up the correct circuit. If we had a bbrtas in the current directory, CIRC would have used that instead.

```
File options:  in=bbrtas out=<stderr> err=<stderr>
Output options:
Displaying:
Reading input from file '/users/mdhutton/mcnc/4/bbrtas.blif' (k=4)
```

The next section of the output file gives basic statistics, as defined in Section 2. Note that there is actually an internally represented "service" (0th) component reported in parenthesis in the component list. This can be ignored.

```
name:   bbrtas
size:   417
edges: 1440
levels: 1
delay: 18
nPI: 4
nPO: 2
nDFF: 7
nLOG: 406
num_unusable: 1
num_unreachable: 0
ncomponents: 1 ( (4) 417 )
```

The degree information comes next. We have the average in-degree of LUTs, average out of LUTs + DFFs + PIs, and each separately; the average and total fanin and fanout by combinational delay level, the max-fanout, nodes with fanout beyond 1 standard deviation of the mean, and larger than 10 in absolute value, and the fanin and fanout vectors as sparse vectors and in full form.

```
avgin_log: 3.55 (0.46)
avgout: 3.47 (10.28)
avgout_dff: 74.43 (12.48)
avgout_pi: 26.00 (6.00)
avgout_log: 2.02 (3.39)
avgin_vec: ( 0.00 3.34 3.58 3.67 3.81 3.22 3.94 3.54 3.00 3.73 3.00
             3.97 3.50 3.00 3.95 3.40 3.80 3.50 4.00 )
avgout_vec: ( 0.00 1.74 1.00 5.33 2.62 3.67 2.83 1.11 6.40 1.73 18.00
              1.24 2.67 12.67 1.10 5.60 1.10 1.00 1.00 )
totin_vec: ( 0 454 283 33 61 58 71 99 15 56 9 115 21 9 83 17 38 14 4 )
totout_vec: ( 625 236 79 48 42 66 51 31 32 26 54 36 16 38 23 28 11 4 1 )
visible_edges: 1449
max_out: 90
high_degree_log: 14
high_degree_pi: 4
high_degree_dff: 7
degree_10plus_log: 17
degree_10plus_pi: 4
degree_10plus_dff: 7
fanin: (0,4) (1,7) (2,34) (3,116) (4,256)
fanout: (1,345) (2,12) (3,6) (4,4) (5,7) (6,9) (8,2) (9,4) (10,1)
        (13,2) (14,5) (15,1) (16,2) (17,3) (20,2) (21,1) (24,1) (29,1)
        (32,2) (53,1) (60,1) (75,1) (76,1) (81,1) (86,1) (90,1)
fanin_vec: ( 4 7 34 116 256 )
fanout_vec: ( 0 345 12 6 4 7 9 0 2 4 1 0 0 2 5 1 2 3 0 0 2 1 0 0 1 0 0 0 0 1
              0 0 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 1
```

```
0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 1 0 0 0 0 1 0 0 0 1 )
```

Next we have sequential information, since bbrtas is a sequential circuit. This information would not be displayed were it purely combinational.

We have the number of bottom nodes (over all levels), the number of invisible nodes (total and broken into DFF, LOG and PO), the number of nodes at each sequential level (sequential shape), the number of back-edges (number of GOs from level 1 to level 0) and their shape-like distribution, and the maximum combinational delay of each level.

```
bot_nodes: 89
invis_nodes: 0
invis_DFF: 0
invis_LOG: 0
invis_PO: 0
seq_shape: ( 304 113 )
back_edges: 325
where-back: ( 0 325 )
level_maxdelays: ( 18 2 )
```

Combinational shape vectors follow. These are over the *entire* circuit, summed. This section is of limited value for sequential circuits.

```
shape      417: ( 11 136 79 9 16 18 18 28 5 15 3 29 6 3 21 5 10 4 1 )
POshape      2: ( . 0 0 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 )
edge-length: ( 0 1035 179 30 35 39 30 25 6 21 6 24 6 4 0 0 0 0 0 )
forward-edge-length: ( 0 889 89 27 25 25 16 14 3 13 3 4 4 3 0 0 0 0 0 )
back-edge-orig: ( 196 116 13 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 )
back-edge-dest: ( 0 67 159 5 11 15 6 24 0 10 0 24 1 0 3 0 0 0 0 )
back-edge-length: ( 0 146 90 3 10 14 14 11 3 8 3 20 2 1 0 0 0 0 0 )
```

Then the vectors for each individual level are presented. The input design has 1 sequential level (beyond the 0th level, the combinational part).

At level 0, we see the number of nodes, GI, GO, PO and shapes of each, as well as the fanout distribution for L0 alone. Note again that the GI/GO counts will seem high, but they count ports/edges not actual nodes. It is quite common, for example, to have a level 1, such as shown, with 120 nodes and 325 ghost outputs.

```
n0 = 308;
L0shape = (4,43,66,9,16,18,18,28,5,15,3,29,6,3,21,5,10,4,1)
nGI0 = 325;
L0GIshape = (67,159,5,11,15,6,24,0,10,0,24,1,0,3,0,0,0,0,0)
nGO0 = 7;
L0GOshape = (0,0,0,0,1,0,1,0,0,1,0,1,0,0,1,0,1,0,1)
nPO0 = 2;
L0POshape = (0,0,2,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0)
nEdges0 = 769;
L0edges = (0,566,66,27,25,25,16,14,3,13,3,4,4,3,0,0,0,0,0)
L0out = (4,250,8,4,3,2,6,1,1,4,1,0,0,2,5,1,2,3,0,0,2,1,0,0,1,
          0,0,0,0,1,0,0,2,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
          0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
          0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0)
```

Similarly for level 1:

```
n1 = 120;
L1shape = (7,93,13,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0)
nGI1 = 0;
L1GIshape = (0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0)
nGO1 = 325;
L1GOshape = (196,116,13,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0)
nPO1 = 0;
L1POshape = (0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0)
nEdges1 = 346;
L1edges = (0,323,23,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0)
L1out = (88,16,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
          0,0,0,0,0,0,0,1,0,1,0,0,0,0,0,0,1,0,0,0,0,0,1,0,0,
          0,1,0,0,0,0,0,0,0,1,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,
          0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0)
```

And, finally, we close with a statement of the time and memory used for the analysis.

```
Circuit Analysis complete.  cpu: 7.21 sec, mem: 888K, time: 8 sec
```

3.3 Using CIRC as input to GEN.

CIRC is also able to output a GEN program in the SYMPLE language which can be then run through GEN to make a "clone" circuit of the original input circuit.

For example, the output of

```
circ in=bbrtas gen
```

appears in bbrtas.gen as the following:

```
/* CIRC 2.2, compiled Mon May 27 14:13:55 PDT 1996.
 */
X = {
    name="bbrtasclone";

    L0 = (@.comb_circ) {      exact=1;
        name="L0"; n=304; kin=4; nPI=4; nDFF=0; level=0; delay=18;
        nBot=1;
        shape=(4,43,66,9,16,18,18,28,5,15,3,29,6,3,21,5,10,4,1);
        nGI=325;        GIshape=(67,159,5,11,15,6,24,0,10,0,24,1,0,3,0,0,0,0,0);
        nGO=7;  GOshape=(0,0,0,0,1,0,1,0,0,1,0,1,0,0,1,0,1,0,1);
        nPO=2;  POshape=(0,0,2,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0);
        nEdges=769;
        edges=(0,566,66,27,25,25,16,14,3,13,3,4,4,3,0,0,0,0,0);
        outs=(4,250,8,4,3,2,6,1,1,4,1,0,0,2,5,1,2,3,0,0,2,1,0,0,1,
            0,0,0,0,1,0,0,2);
        max_out=32; nZeros=4;
    };
```

```
    L1 = (@.comb_circ) {      exact=1;
        name="L1"; n=113; kin=4; nPI=0; nDFF=7; level=1; delay=2;
        nBot=13;
        shape=(7, 93, 13);
        nGI=0;  GIshape=(0, 0, 0);
        nGO=325;        GOshape=(196, 116, 13);
        nPO=0;  POshape=(0, 0, 0);
        nEdges=346;
        edges=(0, 323, 23);
        outs=(88,16,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
            0,0,0,0,0,0,0,1,0,1,0,0,0,0,0,0,1,0,0,0,0,0,1,0,0,
            0,1,0,0,0,0,0,0,0,1,0,1);
        max_out=61; nZeros=88;
    };

    glue=(L0, L1);
};
output(circuit(X));
```

We will discuss the parameterization file further in Section 4 of the document.

4 Using GEN to generate circuits.

GEN has some command-line options, but the vast majority of the parameterization is too complicated to be specified at the command line. GEN is augmented with a rich specification language with which parameter-programs can be written.

4.1 Generating a simple combinational circuit

Thus far we have only talked about how parameters are evaluated in the SYMPLE language. This doesn't actually create a circuit. To have the parameters passed to GEN, and a circuit generated, we use the "output_circuit" command in our script "x.gen" which we invoke with "gen x.gen"

```
    X = comb_circ { n = 500; };
    output(circuit(X));
```

Output from gen is split into two streams. In the stdout stream, the output blif-formatted netlist is printed to x.blif, as specified. The log of information and errors goes to stderr, and is described as follows:

```
  GEN Development Version 2, compiled Thu May 23 09:59:10 PDT 1996.
  Parsing parameters for circuit 'C'
  Random shape with n=500 nTop=9 nBot=2 delay=11 width=182 jumps=5 expand=9.00:

  Generating combinational specs for C
  -- n = 500  nPI=9  nDFF=0  kin = 4  delay = 11  seed=833239250
  --    Shape (  500):   9  60 182 105  57  42  16   7   7   7   6   2
  -- PO Shape (    3):   0   0   0   0   0   0   0   0   0   1   0   2
```

```
-- GO Shape (    0):   0   0   0   0   0   0   0   0   0   0   0   0
-- GI Shape (    0):   0   0   0   0   0   0   0   0   0   0   0   0
-- Edges    ( 1172):   0 950 141  46  29   4   0   2   0   0   0   0
-- Out-degrees (max=18):  3 311 48 44 18 22 16 14 8 2 6 3 0 0 1 1 0 1 2
Building the circuit level-graph
Graph passed steps one and two.  Best was method 3
Splitting nodes to generate the complete circuit-graph.
Degrees fudged: 461, edges fudged 56, edges lost 0 (of 1672 total)
Warning:  Forced to add 1 extra outputs at delay level 11
Warning:  Fixing IO dist'n results in 1 extra nodes, 1 extra outputs.
Graph generated, converting to a circuit.
(Sub)circuit 'C' has been generated.
Circuit generation successful
Elapsed time 3 seconds
```

We have the version and compilation date of the program. Another important parameter is the random number seed (taken from the clock). To get exactly the same circuit again, we should specify "seed=833239250" on the command line.

The defaults.gen file (hence comb.gen) is read for default information, then x.gen is processed. We specified n=500, from which comb_circ specified 9 PIs and the remaining LUTs, with combinational delay 11 and 3 POs). The number of edges was 1172, so the average fanin was about 2.2 (not a particularly dense circuit). Similarly, the combinational delay and distribution of nodes, edges and fanouts are shown. If we run the command line again without specifying the same seed, we will get both a different parameterization and a different circuit. Had we specified the complete parameterization, we would get a different circuit with the same parameterization.

Generating a combinational clone:

To generate a clone of an MCNC (or other) circuit (in xnf), do the following (for example, we use the circuit 5xp1):

```
circ in=5xp1 gen
gen  5xp1.gen
circ in=5xp1clone.blif xnf
```

4.2 Generating a hierarchical or sequential circuit

Sequential circuits are specified in GEN as hierarchical circuits with "glue" ports to combine them together. For example, a finite state machine is viewed as two or more combinational circuits, one of which has primary inputs, and the others of which have DFFs as its primary inputs. GEN will make a sequential circuit in this way by generating the two combinational circuits separately, then gluing them together following a number of rules beyond the current scope of this document.

The user has control over the type of sequential circuit that is generated in the input script. At the simplest level, the user can specify the size of the circuit and the number of I/Os and DFFs and let the rest come from the defaults. For example

```
X = fsm_circ {
```

```
    name = "example5";
    n=500;
};
output(circuit(X));
```

will generate a "fsm-like" circuit with 500 nodes directly from the defaults. On my machine, with seed=834610821, I got a circuit with 6 PIs, 471 nodes, combinational delay 10, 2 POs, 29 DFFs and 145 back-edges (GOs at level 1).

You can also specify the amount of interaction between the levels by giving values for nGI, nDFF and so on. For example

```
X = fsm_circ {
    name = "example2";
    nPI=63; nPO=36;
    nDFF=120;
    n=450+nDFF+nPI;
    kin=4;
    n0=n/2;
    n1=n/2;
    nBack=n/3;
};
output(circuit(X));
```

Above we have specified the number of back-edges in terms of the size of the circuit, and specify the number of DFFs and I/Os exactly. We have asked for 450 LUTs, giving a size 450+nDFF+nPI for the entire circuit.

It is possible to make more difficult hierarchical circuits, but this part of the code is very new, and there will be problems when you try to do it.

For example, see gendir/5-way.gen, which generates 5 separate sequential circuits with a specified number of ghost I/Os, and then glues all 5 together simultaneously.

See also 40K.gen, which generates a large circuit (40000 4-LUTs) from several smaller circuits, with a specified cut-size (for example, to test a partitioner). Here the result is seen as a combination of several state-machines which provide control into a combinational circuit at the next level. By manipulating the parameters it is possible to make a number of different configurations.

Note that the probability of errors increases multiplicatively with the number of circuits in the hierarchy. Whereas there is a 85% or more chance of success at generating a circuit with 5000 nodes, generating 5 such circuits to glue together will only succeed about 44% of the time. This means that multiple runs are often required. However, I have successfully generated circuits with this amount of hierarchy to 150000 4-LUTs within about 10 tries. It is expected that as we refine the parameterization scripts and build more error handling and correction into GEN that this will disappear, and we will be able to generate circuits with a great deal of hierarchy.