Appendix A

Default Parameterization Scripts

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

Whenever GEN is run, the defaults files are automatically read. This sets up the default frames `comb_circ` and `fsm_circ` which are then modified by the user's parameterization. The file "defaults.gen" must exist either in the current directory, or in the directory specified by the GENDIR environment variable.

1 A Brief Introduction to SYMPLE.

SYMPLE is a specification language, as opposed to a programming language. In that sense, it is more like VHDL than like a procedural language such as C. In particular, SYMPLE utilizes lazy evaluation, which means that no concept of temporal or procedural computation exists. Thus, a program such as:

```
x = 5;
output(x);
x = 6;
output(x);
```

is an error because it does not "make sense" to specify the signal x to have two different values. Similarly the statement "x = x + 1;" is an error.

SYMPLE has two main objects: A *cell* is largely analogous to a variable or a parameter, and a *frame* corresponds to a collection of cells. Used appropriately, a frame can also function as a subroutine.

Consider the following SYMPLE program, which covers most of the major concepts in frames:

```
Z = {
    a = 2;
    X = {
        a = 5;
        b = $.a + a;
    };
    out = X.b;
```

```
};
Y = (@.Z) { a = 4;};
output(Y.out);
```

The first thing to note is that *no* evaluation takes place until an **output** statement is parsed. At that point, the cell specified by the output statement must have been defined. To output Y.out, we require the frame Y, evaluated as a copy of frame Z below the top level frame (denoted "@") so Z is cloned (duplicated without evaluation). Then it is specified to modify the value 'a' in the new frame to be a 4 instead of its previous value. Now we can evaluate out, which is X.b. Within frame X, we have a=5 (note: different scope from the 'a' in the parent Y frame), and b = 5 + the parent's 'a' cell (which is 4), giving Y.X.b a value of 9. Thus the output of the program is 9.

The **eval** statement is also a command, which forces a value to be evaluated (fixed with a final value) but not output.

A SYMPLE program is parsed sequentially. As frames are defined as modifiers of previous frames, they are duplicated, and the new values specified. Then the frame is evaluated only when it is output.

Modified frames can function as a subroutine in which all parameters are optional. Note that a guarded parameter evaluation in a SYMPLE "subroutine" frame would look like the following:

```
Z = {
    a = 0;
    X = {
        out = 5 / $.a;
    };
    result = a==0 ? 0 : X.out;
};
V = Z { a = 4; };
output(V.result);
W = Z;
output(W.result);
```

which has the outputs "1.25" and "0."

SYMPLE has a set of library functions which are available to the user. Most of these are used in the obvious way, and the comb.gen defaults file is a good place to look for examples. Some are as follows:

```
/* A is taken from a Gaussian dist'n with mean 1.0546, std. dev .01 */
/* The selected value is then truncated to the range (0,2).         */
a = gauss(1.0546, .01, 0, 2);

/* Here we use standard C if...then expression syntax, and a min function */
IOmax = n < 100 ? n/3 : min(n/2, 600);

/* nearest integer, also floor and ceil, are available */
nEdges1 = nint((n-nIN)*avg_in);

/* the arity of max/min functions is infinite */
```

```
width_lower = max(wlower4a, wlower4b, wlower4c, wlower4d);

/* functions can be nested */
fmax_out = max(lower1, lower2, min(upper1, upper2, sample));

/* exp and log work with natural base */
locality = nint(2 + exp(log(n)/log(10))/exp(2));
```

Subroutines can be created to form data abstraction and hiding. For example, the we can have the following:

```
default_delay = {
    n = $.n;
    delay = nint(1 + gauss(1.2*log(n), 1, 1, n/3));
};
comb_circ = {
    n = 100;
    ...
    delayframe = (@.default_delay);
    delay = delayframe.delay;
    ...
};
```

Here we have defined a frame which evaluates delay, instantiated with a different value *each time we specify a comb_circ*, and taken comb_circ.delay as the default value for delay in comb_circ.

It is quite different to use the statement "delay = (@.default_delay).delay" rather than using the frame modifier, because this would force the delay parameter in the default_delay frame to be permanently evaluated, and we would always get the same Gaussian value from that point on in the execution of the program. In particular, if this was a hierarchical circuit then all sub-circuits sub-circuits would get have the same combinational delay.

Defining a circuit by the statement:

```
X = comb_circ {delay = 4; };
```

means that we have overridden the value of delay to be the *constant* 4 instead of the value evaluated in delayframe. This mechanism allows us to hide intermediate calculations in sub-frames. The easiest way to understand how things are being evaluated is to turn trace on (command-line option "trace") when running GEN, and to play with test scripts.

SYMPLE also has another construct with side-effects. The statement "`in("stuff.gen")`" specifies an include syntax, similar to #include in C. If there is an environment variable GENDIR, symple will look first in the current directory then in GENDIR for the include file.

2 GEN Combinational Defaults File.

```
/*
 *  GEN default script for combinational circuits.
 */

/*  $Revision: 3.0 $ */
```

```
/*  Determine a reasonable nPI, nPO.
 */
default_io = {
    n = -1;

    a = gauss(1.0546, .01, 0, 2);
    b = gauss(0.5627, .20, 0, .62);
    IOmin = 2 * log(n);
    IOmax = n < 100 ? n/3 : min(n/2, 600);
    nIO = nint(min(max(IOmin, exp(a + b * log(n))),IOmax));

    nPI = nint(gauss(1.1 * nIO/2, log(nIO), 2, nIO - 1));
    nPO = (nIO - nPI) > 0 ? (nIO - nPI) : nint(rand(1,nPI));
};


/*  Determine a reasonable delay.
 */
default_delay = {
    n = -1;
    kin = -1;
    nBot = -1;

    /* Have to have enough delay to collect terms */
    mindelay = log(log(n)) + ceil((log(n/nBot))/(log(kin)));

    delay4 = log(log(n)) + gauss(log(n), 1, 1, n/3);
    delay3 = delay4 * log(log(delay4));
    delay2 = delay4 * log(delay4);
    delay0 = kin==2 ? delay2 : kin==3 ? delay3 : delay4;

    delay = nint(max(delay0,mindelay));
};


/*  Determine a reasonable nEdges.
 */
default_num_edges = {
    n = -1;
    kin = -1;

    avg_in = kin==2 ? 2 : 2 + gauss((kin-2)/2, (kin-2)/5, .8, kin-2);
    nEdges1 = (n-nPI)*avg_in;
    lower = 2 * (n-nPI);
    upper = kin * (n-nPI);
    fEdges = min(max(nEdges1, lower), upper);

    nEdges = nint(fEdges);
};

/*  Default edge-length distribution.
 *
 *  Note that we can't do this in the num_edges frame, because we rely
 *  on nEdges as a parameter, and it could have been modified by the caller.
 */
default_edges = {
    n = -1;
    nEdges = -1;
    delay = -1;

    tot_edgelen = delay<=1 ? nEdges : nint(nEdges * (1 + gauss(0,.5,0,.5)));
    nComponents = nEdges;
    veclen = delay;
    nZeros = 0;
    min_nUnits = nint(max(n, nEdges/2));
    min_nMax = 0;
    /* dp_edges = rand(0.75,1.25) ; */
    dp_edges = gauss(1, .5, 0.75,1.25) ;

    edges = exp_dist(nComponents, tot_edgelen, veclen, 0, min_nUnits, min_nMax, dp_edges);
};


/*  Determine a reasonable max_out.
 */
default_max_out = {
    n = -1;
```

```
    kin = -1;
    nEdges = -1;
    nPI = -1;
    nBot = -1;
    delay = -1;

    /* log-linear version */
    a1 = gauss(-1.07181, 0.5, -2, 1);
    b1 = gauss( 0.8242,  0.5, .7, .9);
    sample1 = exp(a1 + b1 * log(n));

    /* linear function of sqrt version */
    a2 = gauss(2, 1, 1, 5);
    sample = a2 * sqrt(n);

    deflate = rand(1,8);
    upper1 = 512/deflate;
    upper2 = n-nPI;
    lower1 = n-nPI == 0 ? 1 : nEdges / (n-nPI);

    /* have a lower bound based on nPI and possible width */
    nInternal = n - nPI - nBot;
    w3a= delay==5 ? nInternal-kin*nBot-kin*kin*nBot-kin*kin*kin*nBot:0;
    w3b= delay==4 ? nInternal-kin*nBot-kin*kin*nBot : 0;
    w3c= delay==3 ? nInternal-kin*nBot : 0;
    w3d= delay==2 ? nInternal : 0;
    w3e= delay >5 ? (2.5 * nInternal) / delay : 0;
    width_upper = max(w3a, w3b, w3c, w3d, w3e);

    lower2 = width_upper / nPI;

    upper = min(upper1, upper2);
    lower = max(lower1, lower2);

    fmax_out = max(lower, min(upper, sample));
    max_out = nint(fmax_out);
};


/*  Determine a reasonable out-degree sequence
 */
default_outs = {
    n = -1;
    nEdges = -1;
    max_out = -1;
    nZeros = -1;

    dp_outs = rand(0.75,1.25) ;
    outs = exp_dist(n, nEdges, max_out, nZeros, 0, 1, dp_outs);
};


/*  Determine reasonable shapes for POs.
 */
default_IOShape = {
    delay = -1;
    nBot = -1;
    nPO = -1;

    minPOBot = nBot;
    maxPOBot = min(nBot, nPO);
    POBot = nint(rand(minPOBot, maxPOBot));

    POSlope = 1;
    POlen = delay;
    POshape = delay==0 ? bi_linear(nPO, 0, 0, 0, 0, 0)
        : bi_linear(nPO, 0, POBot, POlen, delay, POSlope);
};


/*  Determine reasonable maximum width for the shape profile.
 */
default_width = {
    n = -1;
    kin = -1;
    nTop = -1;
    nBot = -1;
    delay = -1;
```

```
    max_out = -1;
    expand = -1;

    nInternal = n - nTop - nBot;
    upper1 = (delay <= 2) ? nInternal : (nInternal / 2);
    upper2 = expand * expand * expand * expand * nTop;
    upper3 = nint(n / (delay / 2));

    upper4a= delay==5 ? nInternal-nTop-kin*nBot-kin*kin*nBot-kin*kin*kin*nBot:0;
    upper4b= delay==4 ? nInternal-nTop-kin*nBot-kin*kin*nBot : 0;
    upper4c= delay==3 ? nInternal-nTop-kin*nBot : 0;
    upper4d= delay==2 ? nInternal : 0;
    upper4e= delay>5 ? n : 0;        /* no upper4 if have high delay */
    upper4 = max(upper4a, upper4b, upper4c, upper4d, upper4e);

    lower1 = nInternal / (delay - 1);
    lower2 = max(nBot, nTop);

    lower = max(lower1, lower2);
    upper = max(lower, min(upper1, upper2, upper3, upper4));

    mean = (lower + (3 * upper)) / 4;
    fwidth = gauss(mean, 2*sqrt(mean), lower, upper);

    width = (delay <=2) ? max(nTop, nBot) : nint(fwidth);
};


/*  Default shape profile.
 */
default_shape = {
    n = -1;
    kin = -1;
    delay = -1;
    nTop = -1;
    nBot = -1;
    jumps = -1;
    expand = -1;
    width = -1;

    shape = rand_shape(n, kin, delay, nTop, nBot, width, jumps, expand);
};


/*
 *  A combinational circuit.  We have the basic set of parameters,
 *  with additions for fanout, shape, edges and output distributions.
 */
comb_circ = {

    name = "C";
    n = 0;
    kin = 4;
    global_max_out  = -1;        /* passed from fsm.gen or above */

    loc1 = log(n)/log(2);
    loc2 = 2 * sqrt(n)/5;
    loc3 = 2 * exp(log(n)/log(10));

    locality = ceil(max(loc1, loc2));

    /* Circuit is complete comb_circ unless overridden */
    level = 0;
    nLatch = 0;

    /* Choose distribution of PI and PO from defaults */
    IOFrame = (@.default_io) { n=$.n; };
    nPI = IOFrame.nPI;
    nPO = IOFrame.nPO;
    nOUT = nPO + nint(3*log(n));
    nIO = nPI + nOUT;

    /* Number of nodes at the last level of the shape profile */
    nBot = nOUT == 1 ? 1 : nint(gauss(nOUT/2, nOUT/2, 1, nOUT));

    /* Choose number of edges from defaults */
    nEdgesFrame = (@.default_num_edges) {
        n=$.n; kin=$.kin; nPI=$.nPI;
```

```
    };
    nEdges = nEdgesFrame.nEdges;

    /* Choose delay from defaults */
    DelayFrame = (@.default_delay) { n=$.n; kin=$.kin; nBot=$.nBot; };
    delay = DelayFrame.delay;

    /* Choose edge-distribution from defaults */
    edgesFrame = (@.default_edges) {
        n=(($.n) - ($.nPI)); nEdges=$.nEdges; delay=$.delay;
    };
    edges = (delay == 0) ? (0) : edgesFrame.edges;

    /* Choose the maximum out-degree */
    MaxOutFrame = (@.default_max_out) {
        n=$.n; kin=$.kin; nEdges=$.nEdges; nPI=$.nPI;
        nBot=$.nBot; delay=$.delay;
    };
    max_out = global_max_out < 1
        ? MaxOutFrame.max_out
        : min(global_max_out, MaxOutFrame.max_out);

    /* Choose out-degree distribution */
    minzeros = max(0, n - nEdges + max_out);
    nZeros = max(minzeros, nint(rand(nBot, nOUT)));
    outsFrame = (@.default_outs) {
        n=$.n; nEdges=$.nEdges; max_out=$.max_out; nZeros=$.nZeros;
    };
    outs = outsFrame.outs;

    /* Choose the shape of I/O things -- PO */
    IOShapeFrame = (@.default_IOShape) {
        delay=$.delay; nBot=$.nBot; nPO=$.nPO;
    };
    POshape = IOShapeFrame.POshape;

    /* Choose the shape profile */
    jumps = delay<=3 ? 0 : nint(rand(0, delay/3));
    _expand = kin * gauss(3.5, sqrt(3.5), 1, 7);
    expand = max(1, min(_expand, max_out/2));
    widthFrame = (@.default_width) {
        n=$.n; kin=$.kin; nTop=$.nPI; nBot=$.nBot; delay=$.delay;
        max_out=$.max_out; expand=$.expand;
    };
    width = widthFrame.width;
    shapeFrame = (@.default_shape) {
        n=$.n; kin=$.kin; delay=$.delay; nTop=$.nPI; nBot=$.nBot;
        jumps=$.jumps; expand=$.expand; width=$.width;
    };
    shape = shapeFrame.shape;
};
```

3 GEN Sequential Defaults File.

```
/*
 *  GEN default script for sequential circuits.
 */

/*  $Revision: 3.0 $ */


/*
 *  Frames to define a FSM circuit, and utility frames to determine
 *  parameters for fsm_circ.
 */


/*  The number of I/Os to a fsm can be smaller than for comb, because of
 *  all of the ghost I/Os.
 */
default_fsm_IOs = {
    n = -1;
```

```
    combIOFrame = (@.default_io) { n=$.n; };
    nPI1 = combIOFrame.nPI;
    nPO1 = combIOFrame.nPO;

    nPImin = max(2, nint(nPI1/4));
    nPImax = nPI1;
    nPI = nint(rand(nPImin, nPImax));

    nPOmin = max(2, nint(nPO1/4));
    nPOmax = nPO1;
    nPO = nint(rand(nPOmin, nPOmax));
};


default_seq_shape = {
    n = -1;
    nDFF = -1;

    /* Choose number of FFs to have */
    _nDFFmin = max(2, n/50);
    _nDFFmax = n/10;
    _nDFFmean = n/20;
    _nDFF0 = nint(gauss(_nDFFmean, sqrt(_nDFFmean), _nDFFmin, _nDFFmax));
    _nDFF = nDFF>0 ? nDFF : _nDFF0;

    n0 = nint(rand(.6, .85) * n);
    n1 = n - n0 + _nDFF;
};



/*
 *  The definition of a generic finite-state machine.
 *
 *  Things supposed to be parameters:
 * name, n, n0, n1, kin, startlevel, max_out, nGI, nGO, nPI, nPO,
 * nDFF, nDFF0, nBack, locality, delay, avg_in.
 */
fsm_circ = {

    name = "S";
    n = 100;
    kin = 4;
    startlevel = 0;
    levels = 1;
    max_out = -1;

    nGI = 0; nGO = 0;
    nDFF = nint(rand(n/20, n/5)); /* usually overridden */

    locality = nint(6 + exp(log(n)/log(10))/exp(2));

    /* Choose an overall max-delay for the circuit */
    fake_nOUT = nPO + nint(min(nGO, 3*log(n)));
    fake_nBot = fake_nOUT == 1 ? 1
      : nint(gauss(fake_nOUT/2, fake_nOUT/2, 1, fake_nOUT));
    delayFrame = (@.default_delay) { n=3*($.n)/2; kin=$.kin; nBot=$.fake_nBot;};
    delay = delayFrame.delay;

    IOFrame = (@.default_fsm_IOs) { n=$.n; };
    nPI = IOFrame.nPI;
    nPO = IOFrame.nPO;

    shapeFrame = (@.default_seq_shape) { n=$.n; nDFF=$.nDFF; };
    n0 = shapeFrame.n0;
    n1 = shapeFrame.n1;

    avg_in_mean = 2*(kin-2)/5;
    avg_in_sd = (kin-2)/5;
    avg_in = kin==2 ? 2 : 2 + gauss(avg_in_mean, avg_in_sd, .5, kin-2.5);
    nEdges1 = (n-nPI-nDFF)*avg_in + nDFF - nGI;
    lower = 2 * (n-nPI-nDFF) - nGI/2;
    upper = kin * (n-nPI-nDFF) - nGI;
    fEdges = min(max(nEdges1, lower), upper);
    nEdges = nint(fEdges);

    /* Will build 2 circuits.  L0, L1 */
```

```
    /* Changed July 30: nBack = nint(rand(n1/20, n0 - nPI)); */
    nBackMean = n0/10;
    nBackStdDev = sqrt(n0);
    nBackMin1 = n1 < 500 ? n1/10 : 2 * log(n1);
    nBackMin2 = delay <= 3 ? n1/kin : 0;
    nBackMin = max(nBackMin1, nBackMin2);
    nBackMax= min(n0 - nPI, 5*n1);
    nBack = nint(gauss(nBackMean, nBackStdDev, nBackMin, nBackMax));

    /* Divide up the edges */
    n1min = (n1-nDFF)*2;
    n0min = (n0-nPI)*2;
    _usable_edges = nEdges - nBack - n0min - n1min;
    L0edgesmax = (n0-nPI)*kin - nBack;
    L1edgesmax = (n1-nDFF)*kin;
    ratio = L1edgesmax  / (L0edgesmax + L1edgesmax);

    _L1edges = _usable_edges < 1 ? n1min
           : n1min + nint(ratio * _usable_edges);
    L1edges = min(_L1edges, L1edgesmax);

    _L0edges = _usable_edges < 1 ? n0min
    : n0min + (_usable_edges - nint(ratio * _usable_edges));
    L0edges = min(_L0edges, L0edgesmax);

    L0delay = delay;
    L0BotMax = min(nDFF + nPO, (n0-nPI)/(delay-1));
    L0BotMin = min(max(nDFF/5, log(n0)), L0BotMax);
    L0Bot = nint(rand(L0BotMin, L0BotMax));
    L0Zeros = nint(rand(L0Bot, L0BotMax));

    _L1delay_min = max(1, log(log(n1-nDFF)));
    _L1delay_max = max(_L1delay_min, min(delay, log(n1-nDFF)));
    _L1delay_mean = (_L1delay_max + _L1delay_min) / 2;
    _L1delay_sd = sqrt(_L1delay_mean);
    _L1delay = gauss(_L1delay_mean, _L1delay_sd, _L1delay_min, _L1delay_max);
    L1delay = ceil(_L1delay);

    L1BotMin = L1delay <= 3 ? nint((n1-nDFF)/kin) : nint(log(n1));
    L1BotMax =  max(L1BotMin, (n1-nDFF)/(2*L1delay));
    L1BotMean = (L1BotMax + L1BotMin) / 2;
    L1Bot1 = nint(gauss(L1BotMean, sqrt(L1BotMean), L1BotMin, L1BotMax));
    L1Bot = min(L1Bot1, nBack);
    L1Zeros = nint(rand(L1Bot, min(L1BotMax,nBack)));
    back_shape = bi_linear(nBack, 0, L1Bot, L1delay, L0delay, 0);

    L0 = (@.comb_circ) {
        name = ($.name) + "L0";
        level = $.startlevel;
nPI = $.nPI;
nPO = $.nPO;
nLatch = $.nDFF;
delay = $.L0delay;
        n = ($.n0);
        kin = ($.kin);
        nGI = ($.nBack);
        nGO = $.nDFF;
nBot = ($.L0Bot);
nZeros = ($.L0Zeros);
        GIshape = $.back_shape;
locality = $.locality;
global_max_out = $.max_out;
nEdges = $.L0edges;
    };

    L1 = (@.comb_circ) {
        name = $.name + "L1";
        level = $.startlevel + 1;
        kin = ($.kin);
delay = $.L1delay;
        n = $.n1;
        nLatch = 0;
   nPI = $.nDFF;
        nPO = 0;
        nGI = 0;
        nGO = $.nBack;
nBot = $.L1Bot;
nZeros = ($.L1Zeros);
```

```
        GOshape = $.back_shape;
locality = $.locality;
global_max_out = $.max_out;
nEdges = $.L1edges;
    };

    glue = (L0,L1);
};
```

4 GEN Special-Circuit Defaults File.

```
/*
 *  GEN default script for "special" circuits.
 */

/*  $Revision: 3.0 $ */

/*
 *  Special types of circuits:
 */

/*
 *  Generates an empty circuit.  The reason we want this is so that we
 *  can have parameterized gluing.  Gluing a null circuit on to any
 *  other circuit does nothing to it.
 */
null_circuit = (@.comb_circ) {
    name = "null";
    level = 0; delay = 0;
    n = 0; nPI = 0; nPO = 0; nIN = 0; nOUT = 0;
    nEdges = 0; max_out = 0;
    nBot = 0; nZeros=0;
    locality = 0;

    junk = kin == 1 ? 0 : bi_linear(0, 0, 0, 0, 0, 0);
    GIshape = junk;
    GOshape = junk;
    POshape = junk;
    shape = junk;
    edges = junk;
    outs = junk;
};


/*
 *  A circuit with nDFF FFs and no nodes.
 *  Expectation is that the user will define a circuit
 *   X = (@.register_file) {nDFF=16; nGO=32;};
 *  and then glue it to the end of something else.  Note that specifying
 *  nGO==nDFF (the default) results in a deterministic circuit -- nDFF FFs
 *  with one GO each.  If nGO < nDFF, an error will ensue during generation.
 */
register_file = (@.comb_circ) {
    name = "RF1";
    level = 1;
    nDFF = 0; /* must override */
    n   = nDFF;
    nIN = nDFF;
    nGO = nDFF;
    nGI = 0;
    delay = 0;
    nEdges = 0;
    nPI = 0;
    nPO = 0;
    max_out = nint(rand(1, nGO-n));

    junk = bi_linear(0, 0, 0, 0, 0, 0);
    GIshape = junk;
    GOshape = junk;
    POshape = junk;
    shape = junk;
    edges = junk;
```

```
    outs = junk;
};
```