Fltsim Detailed Description and Operation

Norm May
Dan Hammerstrom
Oregon Graduate Center
19600 N.W. von Neumann Dr.
Beaverton, Oregon 97006

Technical Report No. CS/E-88-021
May 1988

Abstract

This report is a companion report to CS/E-88-020 and provides a detailed description of the Fltsim program. Detailed information is provided for operation of the fault simulator and for someone to modify the operation of the fault simulator.

TABLE OF CONTENTS

1. FAULT SIMULATOR COMMAND LINE DESCRIPTION

Fltsim reads and writes several files. Each input file contains information required by the simulator to build physical models of the architecture. The output files are used to determine the effects of the faults and as input files for the architecture simulators to simulate operation of the faulted network. The command line for the fault simulator includes several options for specifying source and destination file names. Listed below are the command line optional arguments with the default file names that are used if the option is not specified. All arguments in brackets are optional.

```
fltsim      [-b BIF.in]       [-s fstat.output]      [-T]
            [-t tech.in]      [-o fbif.output]       [-N]
            [-f fault.in]     [-d test.output]       [-h]
            [-p pad.in]
            [-S seed]

Where -T is TEST mode
      -N is No faulted BIF output
      -S is the random # generator seed
      -h is help
```

Calling Fltsim with the `-h` option returns the above text indicating all the options and default file names that can be specified. The `-b` option specifies the mapped BIF file input describing the n-graph, the `-t` is the technology file input information providing device size information, the `-f` is the input fault parameter file to characterize the fault distribution, and the `-p` option specifies the physical architecture data file describing the physical hardware. The `-s` specifies the fault statistics output file to summarize the fault information, `-o` the faulted BIF fields to be used by the architecture simulators, HAS and ANNE, and `-d` the test output file to list intermediate Fltsim values. The default file names are listed inside the brackets. The format and content of these files are described in the Fault Simulator File Formats appendix.

A TEST mode can be specified to write intermediate values of the simulation to a file. TEST mode can be used to troubleshoot problems that occur, or to obtain more statistics on the characteristics of the faults or hardware model. Information such as the calculated size of the PNs, internal block sizes in the PNs, bus sizes and fault locations are written to this file. Also included is the parsed fBIF output, listing information about what the fault simulator thought each line type was (e.g. iotype field or link information.). The test file output is helpful if fltsim returns an execution error.

The TEST mode can be specified two ways. If the `-T` option is specified or if the `-d` option with a file name is specified, the TEST output will be written. If the `-d` option is listed in the command line, the `-T` is not required. The `-T` option writes the test information to the file named test.output. To direct this test output to a different file name, the `-d` option with a file name is specified.

Fltsim normally will produce a unique set of random numbers each time the procedure is invoked. Sometimes it is desirable to have repeatable faults to diagnose problems with input files or Fltsim routines. Repeatable random number generation is accomplished by specifying the `-S` option with the same seed value. The sequence of random numbers generated by the random number generator will be identical for each invocation of Fltsim with the same seed values. The seed value used is printed in the standard output of the fault simulator and is printed in the fault statistics file to allow repeating the execution of any Fltsim run. The seed is contained in the fault statistics file because this file is always generated with each fault

simulation. Given the same input file data, successive executions of Fltsim with identical seed values will produce output files with the same contents as prior executions.

Very large networks can be run though the simulator without generating the fBIF file. The -N option is then used to inhibit the writing of the faulted BIF output. This option does not effect the modeling of faults, and the fault statistics file, fstat, is still produced.

2. FAULT SIMULATOR FILE FORMATS

PAD File

The PAD file describes the physical architecture of the neural network. It consists of sizes of the hardware blocks and the connectivity of the network. Comments are allowed to help describe the network information. Each comment section must be contained on a single line and start with a "/*" and end with a "*/". This is similar to comments in the C programming language, except for the fact that comments cannot span multiple lines. The PAD file has 4 major parts:

- The CN/PN network descriptions
- PTP Network description
- Local PAD region definitions
- PBH Network description

The PAD file starts with a description of the PNs and CNs. The total number of PNs in the network and how many are in the x and y dimensions is listed first. Next is the number of CNs in each PN of the network. Each CN has a number of inputs, which are values from other CNs. These other CN values are stored in a memory for later use. The number of bits in each CN value is identified by the CN_DATA field. Correspondingly, each input has a weight. The number of weight bits in each value is identified by the WEIGHT field. The total number of other CN values stored in any one PN is CN_ENTRY. CN_ENTRY is the total number of input links to all the Sites on a CN as shown in the n-graph. The network uses an addressing scheme where the incoming message's address field is stripped off the message and sent to the address decoder. The address decoder uses a Content Addressable Memory (CAM) to determine the proper CN_MEMORY word to store the message's data field in. A reduction in transmission time and silicon area can be achieved if the number of address bits required is variable. The CN_ADDR field specifies the number of CN entries with the number of address bits used. The total number of CN_ADDR entries should add up to the total number of input links for the CN. An assumption that each CN within the PN has the same address decoder structure was made, allowing the network to be scaled easier. If the number of CNs in a PN is changed, the number of address entries changes by a corresponding amount.

The next block describes the PTP communication network. Each PN is connected via the PTP communication network to all its immediate neighbors. Each message uses handshake lines to communicate, for example, Data Valid and Data Received lines. Message destinations will not have to be the neighboring PN. The message may have to hop several PNs before reaching its final destination. Encoded in each message is the destination PN number and CN address. The destination address and data is multiplexed over a set of data lines. The PTP fields indicate the number of data lines and the number of control lines for each PTP connection. PN connectivity is listed in the PN_CON fields. For the example shown, PN # 2 is connected to PN # 1, 6 and 3. Fltsim requires knowledge of the x,y addresses of the PNs in the network, so the PNs listed in the PN_CON fields must be listed in physical order. That is, the PN at x,y location 0,0 is first, 1,0 next and so on.

The PAD file allows for shorthand notation for grouping regions. Grouping regions gives a specified number of PNs a given region number that can be referred to later. Larger numbers of PNs

```
/*****************************************************************/
/*              PAD VERSION # 6 example file        7/16/87      */
/*              This is an example PAD description file          */
/*****************************************************************/
/* Start with PN/CN descriptions                                 */
/*****************************************************************/
PN: 16;                 /* total number of PNs in the network */
PN_GEO: 4, 4;           /* x,y dimension of PNs */
CN: 2;                  /* number of CNs in each PN */
CN_DATA: 8;             /* represents current state of other CNs */
WEIGHT: 9;              /* # of weight bits */
CN_ENTRY: 7;            /* # entries of other CN states (input links) */
CN_ADDR: 15: 4, 9: 2;  /* # entries: # address lines */
LSM: 2;                 /* # of learning state machines in each PN */
/*****************************************************************/
/* Define Point-to-Point Communication Network                   */
/*****************************************************************/
PTP_DATA: 4;            /* # of PTP data lines between each PN */
PTP_CNTL: 2;            /* # of PTP control lines between each PN */
PN_CON: 0: 1,4;         /* connections between PNs */
PN_CON: 1: 0,2,5;       /* this will be used to map external PN # */
PN_CON: 2: 1,6,3;       /* number system to internal # system */
PN_CON: 3: 2,7;
PN_CON: 4: 0,5,8;       /* CONNECTIVITY FOR ALL PN'S */
PN_CON: 5: 1,4,6,9;
PN_CON: 6: 2,5,7,10;
PN_CON: 7: 3,6,11;
PN_CON: 8: 4,9,12;
PN_CON: 9: 5,8,10,13;
PN_CON: 10: 6,9,11,14;
PN_CON: 11: 7,10,15;
PN_CON: 12: 8,13;
PN_CON: 13: 9,12,14;
PN_CON: 14: 10,13,15;
PN_CON: 15: 11,14;
/*****************************************************************/
/* Define regions of PNs to be used in PBH definition            */
/*****************************************************************/
REGION: 2: 1, 2;                    /* list PNs in region 2 */
REGION: 1: 3, 6, 7;                 /* list PNs in region 1 */
/*****************************************************************/
/* Describe Physical Broadcast Hierarchy Communication Network  */
/*****************************************************************/
PBH_LEVELS: 2;                /* # levels in PBH hierarchy */

PBH_N: 0: 1: 2;         /* PBH PN netlist - includes Region 1 & PN 2 */
PBH_BDCAST: 7;
PBH_DATA: 2,4;          /* 2 data lines in lowest level, 4 in upper 2 */
PBH_CNTL: 3;            /* 3 control lines in all levels */

PBH_N: 1: 2: 3,4;       /* PBH PN net list - Regions 2 & PN 3 & 4 */
PBH_BDCAST: 12;
PBH_DATA: 3,5;
PBH_CNTL: 3,4;
```

Figure B1 - PAD File.

can be more easily developed using these regions.

The last section describes the PBH communication structure. The number of levels in the PBH communication tree is listed first, indicating the number of concentrator and deconcentrator nodes. Next is the information for each PBH Region in the network. Note here that the PBH Region is different from the regions defined earlier in the PAD file. Each PBH Region number is specified in

the PBH_N field, with all the PNs listed for that PBH Region. The PN list starts with all the predefined regions from above listed first, separated by comas, followed by a colon, and then all the individual PNs listed. The number of broadcast data lines, PBH data lines, and control lines is similar. The number of lines for each level of the PBH hierarchy can be specified, allowing differing number of signal lines for each level. All higher levels from the last number listed will have the same number of signal lines. For example, the PBH_DATA of 2,4 indicates that the lowest PBH level has 2 data lines, the second and all higher levels have 4 data lines.

With all the information given in the PAD file, a block diagram of the physical hardware is possible. The block diagram will contain sizes of the hardware blocks internal to each PN and all the connectivity between the PNs.

Technology File

The technology file describes the sizes of the individual elements, such as memory cells or buffer sizes. Each parameter is specified by a keyword followed by a numeric value indicating the size. Only one parameter is allowed per line and the text after the numeric value can be used as a comment. Entire lines may be used for comments and have to be preceded with a "/*".

The values in the technology file are multiplied by the dimensions given in the PAD file to determine the silicon areas required to implement the functions. The sizes for the components must be given in microns and square microns. The memory_size indicates the size for a single bit memory to store data in the CN_MEMORY block containing other CN values. The adc_size and dac_size are for the analog arithmetic unit to calculate the CN output value. The buffer_size is for the PTP and PBH data and control buffers to drive the bus to the concentrator nodes. The cntl_size indicates the amount of global control circuitry for a PN. The line_width is the line width for the PTP and PBH bus lines only. Line width is used to determine the total amount of silicon area a bus line occupies. Address decoding within a PN is done using a Content Addressable Memory cell. The size of the CAM cell is listed as the addr_dec_size field. Lsm_size is a scale factor for the amount the LSM block increases for each independent learning algorithm used in each PN. Weight_size indicates the memory bit size to store each bit of the weight field. Each PN has a PTP and PBH demultiplexer and control section. This section determines if it is the final destination for the message by examining the PN destination number. Once the message reaches the final PN destination, this section separates the address field and the data field and sends them to their correct destination, the address decoder and CN MEMORY. The size of this section is shown in the ptp_demux_size and pbh_demux_size fields.

```
                    /* file:  technology file */
                       /* date:  27 Nov 87 */
memory_size          50   /* memory size */
adc_size            750   /* adc size */
buffer_size         100   /* buffer size */
cntl_size           800   /* size of control section */
line_width            2   /* line width */
dac_size            750   /* dac size */
addr_dec_size        75   /* address decode size */
lsm_size            960   /* learning state machine size */
weight_size          50   /* memory cell size for weight storage */
ptp_demux_size      100   /* ptp control, addr compare, & demux size */
pbh_demux_size      100   /* pbh control and demultiplexer size */
```

Figure B2 - Technology File.

Fault Parameter File

The fault parameter file includes all the parameters to be used in the fault generation routine. Each parameter is specified by a keyword followed by a numeric value indicating the size. Only one parameter is allowed per line and the text after the numeric value can be used as a comment. Entire lines may be used for comments and have to be preceded with a "/*". Fault parameters are used to generate the locations of the faults and the types of faults.

```
                    /* file:  fault file */
                    /* date:  27 Nov 87 */
defect_den   15     /* average defect density per sq inch */
s-a-0         0.30  /* fraction of open faults */
cluster       0.3   /* clustering coefficient between areas */
in_out        0.40  /* inner/outer fault density ratio */
```

Figure B3 - Fault Parameter File.

The average defect density for the simulation is shown by the field defect_den. The defect density is multiplied by the area of the network to determine the average number of defects to place. The fraction of Stuck-at-0 faults is a parameter, which also determines the fraction of Stuck-at-1 faults. Fault clustering can be varied by the cluster parameter. A typical value for the fault clustering coefficient for a wafer is around 0.3. The fault radial distribution is determined by the in_out parameter. The in_out parameter is the ratio of the defect density of the inner area divided by the defect density of the outside area. To simulate a wafer, the ratio of the inner area defect density to the outer area defect density will be less than 1. To simulate faults on a die in a wafer, the ratio should be set to 1, as the radial distribution only applies to the wafer model.

BIF File

The BIF file describes the n-graph of a neural network, which is described in more detail in Casey Bahr's Thesis, "ANNE: Another Neural Network Emulator" (OGC). The BIF file required by the fault simulator must be mapped to the physical implementation of the network. The mapper routine should be used to assign n-graph nodes the p-graph nodes needed by the fault simulator. BIF describes the n-graph Connection Node connectivity. The BIF lists Group information about the CNs, followed by a list of CNs, a list of Sites for each CN, and a list of links for each Site. Each list describes a different portion of the connections of the n-graph.

Fault Statistics

The fault statistics file contains the fault statistics of the faults in the network. The statistics show where faults occurred and what effects on the n-graph they had. When simulating the network using HAS or ANNE the statistics will provide information to aid in understanding the modified circuit operation. Figures B4 and B5 shows a portion of the fstat file.

```
fault statistics
run time = Mon Dec 21 11:42:37 1987

fault # 0
 DATA    S_A_0 in dac
fault index = 1
fault modifier = 0
fault input site
fault pn x,y = 0,0 which is PN # 0
fault offset = 182

fault # 1
   ALL VALUES IN THIS PN,DO NOT CHANGE DATA ADDR S_A_0 in pncntl
fault index = 195
fault modifier = 0xffffffff
fault output link
fault pn x,y = 1,1 which is PN # 3
fault offset = 0

*** END OF LIST FOR SINGLE FAULTS ***
*** FAULT COMBINATIONS LIST ***
   (determined when reading BIF file)
   (fault #'s correspond to faults listed above
    and the fault order below is the order faults
    are placed in the fBIF file)
CN index = 0
Site index = 0
Link offset = 0
weight (link) offset = 0
Site index = 1
Link offset = 0
weight (link) offset = 0
Link offset = 1
        *** New worst fault #1
weight (link) offset = 1
Link offset = 2
        *** New worst fault #1
weight (link) offset = 2
Link offset = 3
weight (link) offset = 3
weight (link) offset = 0

BIF file statistics
section         number          faulted         percent faulted
CN              12              1               8.33
SITES           24              0               0.00
LINKS           72              18              25.00
WEIGHTS 72              8               11.11
```

Figure B4 - Fault Statistics File.

```
BIF utilization of the Hardware defined in the PAD file
PAD:    16 CN's
        2 Sites (For now, max sites from BIF file)
        2048 Links
PN(0,0):
        2 CN's          12.50 percent utilization
        2 Sites's       100.00 percent utilization
        5 Input Links   0.24 percent utilization
PN(0,1):
        2 CN's          12.50 percent utilization
        2 Sites's       100.00 percent utilization
        8 Input Links   0.39 percent utilization
PN(0,2):
        2 CN's          12.50 percent utilization
        2 Sites's       100.00 percent utilization
        5 Input Links   0.24 percent utilization
PN(0,3):
        0 CN's          0.00 percent utilization
        0 Sites's       0.00 percent utilization
        0 Input Links   0.00 percent utilization
PN(1,0):
        2 CN's          12.50 percent utilization
        2 Sites's       100.00 percent utilization
        8 Input Links   0.39 percent utilization
PN(1,1):
        2 CN's          12.50 percent utilization
        2 Sites's       100.00 percent utilization
        8 Input Links   0.39 percent utilization
PN(1,2):
        2 CN's          12.50 percent utilization
        2 Sites's       100.00 percent utilization
        2 Input Links   0.10 percent utilization
PN(1,3):
        0 CN's          0.00 percent utilization
        0 Sites's       0.00 percent utilization
        0 Input Links   0.00 percent utilization
```

Figure B5 - Fault Statistics File (con'd).

The fault statistics are grouped into five sections:

- The header
- Hardware faults
- BIF faults
- BIF fault statistics
- BIF utilization

The fstat header consists of a description of what is contained in the file, "fault statistics", and the time the Fltsim program was executed. All the output files from Fltsim contain a time stamp, which will be identical for all files generated from a single execution of Fltsim.

The next section lists each fault placed in the hardware circuitry with information describing the specific location and how it will effect the operation of the circuit. For example, fault #1 is a fault in the PN control hardware block of PN #3 with a x,y location of (1,1). This fault will be modeled by not allowing the output data links to change value. The actual fault index and modifier to be put into the fBIF file is listed, which indicates to the fault routines, how to model the fault. The fault offset indicates an offset within the hardware block that the fault occurred in, and is specific to the hardware

block. See the Fault Effects appendix for more information of how to interpret the offset value.

The BIF faults list which hardware faults actually effect the operation of the n-graph. Two types of entries are used, "New worst fault" and "combine for new worst fault". "New worst fault" indicates the hardware fault listed has more impact on the operation of the network than the previous fault, or it is listed for the first fault to effect a BIF section. "Combine for new worst fault" indicates that the listed fault has been combined with the previous worst fault.

The BIF fault statistics summarize the faults placed in the BIF file. The total units for each section and the number and percentage of faulted units are listed.

The last section, BIF utilization, shows the utilization of the hardware circuitry by the BIF description. The number of sections available in the hardware (derived from the PAD file) is listed followed by the utilizations of the BIF sections for each PN. Extreme underutilization of the hardware by the BIF file may lead to invalid conclusions to be made. For example, two networks that use the same PAD file are simulated, but one has a 50% utilization of the hardware and the other has a 100% utilization. Both circuits use the same PAD file, so the size of the hardware will be the same, resulting in the same average number of physical faults. Since only 50% of the hardware is used in the first network, a higher portion of the faults will be in unused hardware, which will not affect the n-graph, leading to a false sense of fault tolerance.

Faulted BIF

The faulted BIF file (fBIF), contains the fault information to be included by the architecture simulator fault routines. Arrays of fault information will be initialized to be accessed by the fault routines which are used to modify intermediate CN node calculations. An example if the fBIF file is shown in Figure B6.

The beginning of the fBIF file contains a comment section providing information about the generation of the fBIF file. The source BIF file used to generate the fBIF file is listed to coordinate the proper use of the same BIF file and fBIF file by the architecture simulator. Using the fBIf file with a different BIF file other than the one used to generate the fBIF file will cause unpredictable results.

Four arrays are initialized in fBIF, flts[], fcn_ptr[], fsite_ptr[][], and flink_num[][]. Flts[] is an array of fault indexes and modifiers to be used to determine how to modify the interface CN value. The other three arrays are indexes into the fault array, used by the fault routine to access the appropriate fault index and modifier. Fcn_ptr[cn_index] points to the CN fault data in the flts[] array. Fsite_ptr[cn_index][site_index] points to the site fault in the flts[] array. Flink_num[cn_index][site_index] contains the number of links for each site and is used to ensure that the link offset is valid. The link fault index into the flts[] array is calculated as an offset from the site fault index. For example, the first link fault value for each site is the first number after the site fault value. The weight fault index is next followed by the link fault value for the second link.

Test Output

The test file output contains intermediate values used in the fault simulation process. Information about the sizes of the calculated physical network and its connectivity, parsed BIF output, fault location information and CN/PN mappings are included in the test file output. The test file can be used to verify operation of the simulator and to debug problems with parsing files, or other problems. Due to the length and variety of information contained in the test file, it is not listed here.

```
/*********************************************************
 *
 * fbif file:  FAULT INDEXES AND MODIFIERS
 *
 *      source bif file = FF4.bif.0has.in
 *      run time = Mon Dec 21 11:42:37 1987
 *********************************************************/
static struct flts {
              unsigned short  index;
              unsigned int    mod;
} flt[] ={
0,0,
0,0,
195,-1,
0,0,
195,-1,
0,0,
0,0,
17,-513,
0,0};

static int fcn_ptr[] = {
0,
13,
26,
39,
52,
71,
109,
128,
167};

static int fsite_ptr[128][2] = {
{1,4},
{14,17},
{27,30},
{40,43},
{53,62},
{72,81},
{-1,-1},
{-1,-1},
{-1,-1}};

static int flink_num[128][2] = {
{1,4},
{1,4},
{4,4},
{4,1},
{0,0},
{0,0},
{0,0}};
```

Figure B6 - Faulted BIF File.

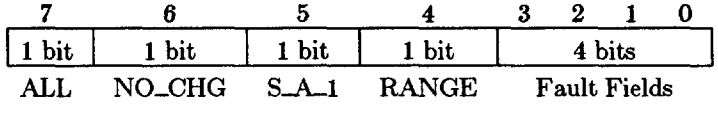
3. ARCHITECTURE SIMULATOR TO FAULT SIMULATOR INTERFACE

The fault information that is to be modeled by the architecture simulators, HAS and ANNE, is conveyed though the fBIF file. Fault routines are called from the architecture simulator routines and user's routines to model the fault's operation. The process of calling these routines, and the information to be be conveyed to them will be described in this section. More detail about this interface is provided in the HAS and ANNE architecture simulator descriptions.

Three files are used by Fltsim to model faults by the architecture simulator: fault.h, fault.c and fbif.output. Fault.h defines constants used by the fault.c routines and Fltsim. Fault.c contains the subroutines to model faults in different n-graph sections. And fbif.output sets an array of fault fields for each subsection of the BIF file. This array of fault fields consists of a fault index and modifier for each CN, each CN Site, and each Site Link and Weight. In order for the architecture simulators to call the fault routines and access the fault fields, certain requirements must be satisfied. An include statement in fault.c includes the fBIF fault fields, which must correspond to the fBIF file produced by Fltsim to model the faults. Also, the fBIF file must have been generated from the same BIF file as being used in the architecture simulator. The fault routines need to be linked and loaded with the user's fault routines. Each time the fBIF output changes, the fault.c file must be recompiled.

Each fault field in fBIF consists of two numbers, a fault index and a fault modifier. The fault index uses 8 bits to indicate the type of fault and where to model the fault. The file fault.h defines constants for the fault fields. Since the fault index value is also used internally to Fltsim, not all the bits will be used by the fault routines. All the bits will be defined here for completeness. Bits 0-3 are the fault locations that indicate if the fault was a data word fault or an addressing fault. Data word faults either modify the targeted value using bit operations, or do not update the value at all. Address faults change the routing of the network, so that the address that the message is being sent from/to will be altered. Bit 4 is set if a range of target values are to be faulted. Bit 4 is not used by the fault routines. Bit 5 indicates if the fault is a Stuck-at-1 (bit 5 high) or a Stuck-at-0 (bit 5 low) fault. Bit 6 if set high indicates that the target value should not be altered. That is, an old value and a new value are both passed to the fault routine, where if the NO CHANGE bit (bit 6) is asserted, the value returned is the old value. For example, the NO CHANGE fault is used when a handshake line is damaged and the destination node does not receive new values from the source node it is supposed to be connected to. The input to the site would always stay at the same value. Bit 7 indicates that all the CNs in a given PN are faulted when set high. Bit 7 is not used by the fault routines.

Fault Index



The NO CHANGE bit has the highest precedence in the fault index, so if it is set, the old value is returned, and the fault modifier is not used. Otherwise, the Stuck-At bit is used to determine the operation of the routine. For a data word fault, the Stuck-At bit will determine if the new value passed should be AND'ed with the fault modifier (S-A-0) or OR'ed with the fault modifier (S-A-1). The new value is so modified, and the routine is finished. If the fault is an address fault, the address routing is modified. Note that both the address and data can be corrupted. In this case, the fault modifier will be used to modify both the address and data fields.

The general steps for calling the fault routines are as follows:

1. For each input link:
   a. Fault the input link and corresponding weight. ..... `flt_lkwt()`
2. Update old input link values.
3. Calculate all the site functions for the CN.
4. Fault all site function outputs. ..... `flt_site()`
5. Calculate the CN function.
6. Fault the CN function output. ..... `flt_cn()`
7. For each output link:
   a. Fault the data and the destination address. ..... `flt_olink()`
   b. Send the output to the next CN address
8. Calculate the weight functions
9. Fault the weight values. ..... `flt_wt()`
10. Update the old weight values.

These steps are performed for each CN calculation in the network. For step 1, faulting all the input links and weights to the CN, a link and weight fault routine, `flt_lkwt()`, is called for each input link to the CN. The fault routine will return the value to use as the link input and weight depending on the fault fields listed in the fBIF file. Figure C1 summarizes the parameters for all the fault routine calls. Six arguments are passed to the link/weight fault routine, the current CN index and site name that this link attaches to, the link index, the old (or previous) value of the link input, a pointer to the new link input and a pointer to the weight for the link input. After the subroutine returns, the contents of the pointer to the new link input will contain the value to use as the link input and the contents of the pointer to the weight will contain the new weight. Both the old value of the previous link input and the current link input are passed to the fault routine to model faults where the input link is defective, not allowing new inputs to be transmitted. Faulty input links which do not allow transfers to take place will set the new input equal to the old input value. The initial old value will need to be determined for the first call for each link input. After the link fault routine completes, the old value can be set to the current value for the next pass.

The site function for each site of the CN is calculated in step 3. The output of each site function is faulted in step 4. The site output is faulted by calling the site fault routine, `flt_site()`, with the current CN index and site name, and a pointer to the site output value. The fault routine will calculate the new site value, using the fault fields, to be returned as the contents of the site output pointer.

Faulting the CN function is done similarly. The CN function calculates a new value and calls the CN fault routine, `flt_cn()`, passing it the CN index and a pointer to the new CN value. The fault fields are accessed to modify the CN output.

The output links are faulted by calling `flt_olink()`, which modifies the destination address and data value to send. The CN output would normally be sent to the destination CNs using the output links. Pointers to the destination CN address and the CN output are passed to a fault output link routine along with the current CN index, site name and link index. The address and CN output are modified using the fault fields in the fBIF file to change the routing and the CN output. The modified CN output is then sent to the destination CN using the modified routing.

Steps 8 through 10 are used for networks with dynamic weights that are calculated during execution of the network. Step 8 calculates the new weights for all the input links using a selected learning algorithm. Step 9 is to fault the weights by calling a fault weight routine, `flt_wt()`, passing the CN index, site name, link index, the old weight value, and a pointer to the new weight value. The fault fields in the fBIF file are used to potentially modify the contents of the new weight value. Some faults may cause the new weight not to be calculated or saved

```
flt_lkwt(cnindex,site,link,old_link,new_link,weight)
  int   cnindex;        /* current cn index */
  short site;           /* current site name */
  short link;           /* current link index */
  short old_link;       /* previous link value */
  short *new_link;      /* new link value */
  short *weight;        /* weight for link */

flt_site(cnindex,site,new_site)
  int   cnindex;        /* current cn index */
  short site;           /* current site name */
  short *new_site;      /* pointer to the new site value */

flt_wt(cnindex,site,link,old_weight,new_weight)
  int   cnindex;        /* current cn index */
  short site;           /* current site name */
  short link;           /* current link index */
  short old_weight;     /* previous weight value */
  short *new_weight;    /* new weight value */

flt_cn(cnindex,new_cn)
  int   cnindex;        /* current cn index */
  short *new_cn;        /* new cn value */


flt_olink(cnindex,site,link,new_addr,new_data)
  int   cnindex;        /* current cn index */
  short site;           /* current site name */
  short link;           /* current link index */
  short *new_addr;      /* new address */
  short *new_data;      /* new output link value */
```

Figure C1 - Fault routine parameters.

in memory, resulting in the weight never being updated. The old weight is used in this case, where the new value is set to the old value. After the fault weight routine completes, the old value is saved for the next pass. The weights are faulted twice, once with the input links and once when the weight values are updated. There are two reasons for doing the weight fault twice. First, steps 8-10 are optional for networks that do not calculate new weight values. Second, the weight values are accessed by the CN at two different times so the faulted value is required twice. So for consistency between dynamic weight networks and static weight networks, the weights are faulted twice. Faulting a value twice has no adverse effects, since the same fault index and modifier are accessed both times. And faulting a faulted value does not change the value. For example, AND'ing the fault modifier with a value a second time does not alter the number.

The actual code to implement the fault routines will depend upon the simulator used and the site/CN functions and data structures used. Some of the fault routines may be called from the architecture simulator, and hence should not be called by the user's routines. With the general guidelines presented here, the interface of the fault routines to the architecture simulator routines and the user's routines can be implemented. The specific architecture simulator description should describe which fault routines are to be called by the user's routines.

Shown below is a simplified section of the user's routine for HAS with the fault routine calls shown in bold print:

```
if(userfx_mode == 1)
   { /* receive link input messages */
   L->old_inval = L->inval;    /* save old value for fault routine */
   L->inval = mes_value;       /* receive message input to node */
   }
else if(userfx_mode == 2)
   { /* Site function - Sum of Products (called once per site input) */
   siteval = S->siteval;
   wt = L->inval;
  flt_lkwt(C->cnindex,S->sitename,L->link_index,
            L->old_inval,&(L->inval),&wt);
      /* fault input link and weight */
   inval = inval * wt;
   siteval += inval;
   S->siteval = siteval;
   }
else if(userfx_mode == 3)
   { /* CN function - Sigmoid function */
   siteval = S->siteval;
  flt_site(C->cnindex,S->sitename,&siteval);      /* fault site output */
   C->output = (1/(1 + exp(-1.0 * siteval)));
   }
else if(userfx_mode == 4)
   { /* send outputs to next CN's */
  flt_olink(C->cnindex,S->sitename,L->link_index,
               &(C->index),&(C->output));
      /* fault destination address and CN output */
   sprintf(buf,"%d %d %d",C->index,C->output,time);
   send_output(C->index);
   }
```

Since the example shown does not use dynamic weights, step 8-10 are omitted; the `flt_weight()` routine is not called.

4. FAULT EFFECTS

The induced faults reflect the actual physical faults that will occur in the system. Some approximations were used in the fault model to simplify the design of the fault simulation and because the CAP network architecture is still in the design phase. This section will cover each section of the hardware block diagram, describing the general function of the block, what types of faults can occur in the block, how the faults effect the function, and how the faults will be modeled in the n-graph. The n-graph is used by the architecture simulators to simulate the operation of the network. The fault fields written in the fBIF file by Fltsim modify the n-graph operation. The hardware blocks described here, except for the bus structures, are replicated for each PN in the network. Potentially several CNs will be in each PN as described earlier in the paper. Some faults effect only one CN in the PN, whereas other faults will effect all the CNs in the PN.

There are three basic types of faults modeled here, Stuck-At-1, Stuck-At-0, and NO CHANGE. The S-A-1 and S-A-0 faults OR or AND a fault modifier with the value, forcing bits in the word to be always high or low. The NO CHANGE fault forces the value not to be updated, that

is, it retains its previous value.

CN MEMORY

The CN MEMORY section is a RAM that stores values of other CN's function outputs or states. The CN MEMORY words stored here will be inputs that the CN function will use to calculate its output. There is one word for each input link to the CN. Each input link gets another CN's output to write a unique word in the CN MEMORY. Only those CNs connected to this CN store their values in this CN's memory.

The most common defect in RAM structures is to lose single bits in the stored data words. When a value is read from the RAM, a bit in the stored value will be always have a high or low value, regardless of what value was written. Defective bits could be caused in the physical hardware by shorts, opens or leakage current between cells. Faults not modeled by Fltsim are RAM control structure faults. Control structure faults will cause multiple faults in the RAM with only one fault in the device. Row or address decoding faults for the RAM are examples of control structures that could be defective.

Defective bits in the memory will change the input link value for a specific link. A random bit in the memory is chosen to be stuck high or low. The defective bit is mapped to the word the fault occurred in. The defective word corresponds to a specific input link. This input link in the n-graph will be faulted. The corresponding link will be determined by the listed order of the links in the BIF file. The first link listed in the BIF file for a particular PN will be the first word in the memory, the second link in the PN will be the second word in the memory, etc.

ADDRESS DECODER

The address decoder uses a Content Addressable Memory (CAM) structure to match the incoming CN address with the corresponding CN value in the CN MEMORY. The incoming CN value is stored in the CN memory word that has a matching address. If no address match is found, the value is not accepted by any of the CNs in the PN. The connectivity of the network is stored in the CAM that is described by the link section in the BIF file. To save silicon area, the addresses can be of variable length. The more common local addresses can be encoded with fewer address bits.

Faults in the address decoder will cause improper decoding of the CN address, resulting in CN data values being written to wrong memory words (incorrect links), no value being written, or multiple CN values being written at once. Faults in the global control of the CAM may cause entire rows or columns to malfunction. Single bit faults are the only faults currently considered by Fltsim.

To fault the address decoder, a random bit within the address decoder is chosen to be stuck high or low. The bad bit is located within the addresses stored in the CAM. Since variable length addresses are allowed, each different address length section has a different probability of a faulty address bit. The faulty address word and faulty bit position are identified. The address word is an offset within the address words for a PN. There is one address word per input link for each CN in the PN. The order of the CN address words (for each input link) corresponds to the order listed in the BIF file. The link address for the faulty address order is listed in the BIF file. The link address for the faulty address word is modified to have either a stuck high or stuck low bit. In the architecture simulators, the destination address will be modified so that if the new address matches some PN/CN in the network, the message will be routed to it. In most cases, the new address will not match a PN/CN, and the message will be lost.

WEIGHTS

The WEIGHTS section stores all the current weights for each CN input link to be used in the site function. One weight word is used for each input link. The Weights section is a RAM that is addressed either when the site function is to calculated, or when the Learning State Machine is to read or update the value. Typically the weight is multiplied by the input link value in the site function.

Since the weights section is a RAM, faults here are similar to the faults in the CN MEMORY section. Any weight bit could be stuck, and would always read a 1 or a 0. Stuck bits will effect the site calculation and the weight update or learning algorithm. To model weight faults in the site calculation, the weight value is modified with either stuck high or stuck low bits before using the value in the site function. To model faults while calculating the new weight using the learning algorithm, the new weight value is faulted with the stuck bit after the new weight is calculated. That is, the weight was modified and stored as the current weight when the site function was called. The LSM used this faulted value to calculate the new weight, and that weight was faulted again, since it was stored in the same RAM word with the same fault. The second time the weight is modified, faults in the LSM are also taken into account.

LEARNING STATE MACHINE

The Learning State Machine (LSM) is a PLA running as a background task that samples and updates all the weights for the site function according to a predefined learning algorithm. Various inputs are used to calculate a new weight value, and may include the current weight, the CN function output, and the CN link input. If any of the inputs to the LSM are faulted, the new weight value will be faulted. All the LSM input faults are modeled by the other sections in the PN. Only internal faults to the LSM are considered in this block. LSM faults could cause the LSM not to work at all or to update the weights with incorrect values. If the LSM does not work at all, none of the weights for the PN will be updated with the new weight value, otherwise the fault will cause the new weight values to have a stuck bit. A predetermined fraction of faults will cause the LSM not to function at all. If a fault occurs in one of the LSMs, all the CNs updated by that LSM will have the same fault in all the weight values. When a fault occurs in one of multiple LSMs in a PN, 1/(# LSMs) of the weights are faulted. See the WEIGHT section for more details of how LSM faults are modeled.

DAC

The Digital to Analog converter (DAC) will calculate the site functions for the CNs in the process of converting the digital words to analog signals. Corresponding words from the CN MEMORY and the WEIGHTS sections are input to the DAC, converting the digital signals to analog signals while performing the site function simultaneously. The analog output represents the output of the site function.

Faults in the DAC section will cause the DAC output to be at an incorrect level, resulting in an incorrect site function output. Incorrect output levels can be caused by any one of the input bits being stuck, which causes a fault identical to a bad bit in the CN MEMORY. Since the faults to the inputs are the same, input bit faults are modeled in the CN MEMORY and WEIGHTS sections. Also, the output of the DAC could be stuck, causing its output to be always stuck at one of the supply rails, which is modeled as the output of the site function to always being all ones or all zeros.

ADC and MUX

The Analog to Digital Converter (ADC) will perform the CN function calculation with analog signals. The ADC will combine all the analog signals from the outputs of the DACs into one analog signal to be converted back to a digital word that represents the CN function output to be sent to other CNs. There is one ADC section for each CN in the PN.

Faults in the ADC section will cause the CN output value to be corrupted. Faults in the inputs to the ADC will be modeled in the DAC hardware block by changing the output of the site function. The analog CN function output signal could be faulty, resulting in the digital CN function output being all ones or all zeros. The faulted CN output is sent via the output link to other CNs in the network.

CONTROL

The Control section groups together the global control circuitry for the PN. This section will not necessarily be in one physical location in the PN, but will be spread over the entire PN circuit. Clock signals that pace the PN circuitry and enable and disable various functions will be the type of control signals that will be included in this section. Any faults on these control lines will have a major impact on the operation of all the CNs in the PN. So faults here simply disable the whole PN. None of the output links from the PN change state, resulting in a static PN state.

PTP BUS

The PTP communication bus lines transfer messages between PNs/CNs. The bus lines are driven by the PTP DATA/ADDRESS/CONTROL buffers. Four sets of bus lines are present for each PN to connect to each neighboring PN. Messages may travel through several PNs before reaching their destination.

Faults may occur in the bus in the form of opens or shorts. Shorts will likely connect two adjacent runs by excess material in fabrication. Opens will occur if a defect causes a discontinuity in the signal path. Shorts are modeled as stuck high and opens as stuck low signals in Fltsim. The control lines that handshake the data transfers may be faulted causing no data transfers to occur. As with the PTP Buffers, the entire message path is checked for faults. Any faults found in the message path will be modeled in the output link of the source CN. If any of the data/address lines are stuck, the output of the CN function will be altered by modifying bits in the message. If the control lines are stuck, the transfer will be completely disconnected.

PTP DATA/ADDR/CNTL BUFFERS

The PTP DATA/ADDR/CNTL BUFFERS are the interface for the PTP communication network. This section includes bidirectional buffers for each data/address line to send and receive the messages, and buffers to control the handshake of the data. The size of these buffers will be larger to drive the capacitance in the data bus, which will make them more prone to faults. Four sets of PTP buffers are used for each PN to send/receive messages from/to any neighboring PN.

Faults may occur in any of the four sets of buffers. Faults in the buffers will effect all the CNs communicating through this set of buffers. The CN does not have to be local to the PN. Messages may travel through several PNs before reaching the destination PN(CN). If any of the intermediate PTP buffers are faulty, the message will be corrupted. The message path is determined by traveling in the x direction first, then in the y direction. For the source and destination PNs, only one side is checked for faults. Intermediate PNs have two sides checked, as the message enters one side and exits on another. Any faults in the message path are modeled at the output link of the source CN. If the output buffer for any of the data/address signals is faulted, the output of the CN function will be modified. If the input buffer for any data/address signals is faulted, the output of the sending CN will be faulted. For data/address faults, bits are either stuck high or low. Since the data is multiplexed, several bits will be stuck. Fltsim does not distinguish between faults on input buffers or output buffers. If any of the control buffers for the handshake are faulted, all transfers of data using these buffers will be impaired. PTP control signal faults are modeled by not updating the input link values.

PTP CONTROL/DEMUX

The PTP Control/Demux hardware block controls the input/output operations of the PTP interface for the PN. Messages are received from the four sets of PTP buffers. The messages are multiplexed, so this unit reassembles the subwords into a message unit. The PN destination portion of the address field is checked to see if this PN is the destination. If this PN is not the destination, one of the four PTP buffers is selected to forward the message, and the message is broken into subwords to be multiplexed over the data lines. If the PN is the destination, the address field is sent to the address decoder and the data field to the CN MEMORY. For output operations, the CN function output from the ADC is sent to the PTP control section. The destination address is added to the message

and the message is multiplexed over one of the four PTP communication buses.

Faults in the PTP control will corrupt the PTP message. Potentially, faults in the PTP Control could cause either single bit faults in the message, or all the bits in the message to be faulted. More commonly, faults in this section will cause the whole section to malfunction, inhibiting all PTP communication for the PN. The individual stuck bits are accounted for in other hardware sections. Faults in the PTP Control will disable all the PTP transfers for this PN. All PTP messages routed through this PN will be disabled, and the fault modeled at the source PN PTP output link.

PBH TRANSMITTER BUS

Several separate PBH regions may exist in the network. Each operates and is modeled as a separate autonomous bus structure. Each PBH region has a binary tree structure to send messages to a top concentrator node via the PBH TRANSMITTER BUS. Then the message is sent over to a separate binary structured RECEIVER BUS to broadcast the message to all the PNs in the region. The PBH TRANSMITTER BUS contains the multiplexed data/address lines, and some handshake control lines to send the message to the top concentrator node.

Faults in the bus, as with the PTP BUS, can cause opens or shorts in these signals. The shorts and opens are modeled by assigning a stuck high or stuck low fault to one of the data signal lines in a particular level of the PBH tree. This will cause several faults in the address and data fields due to the multiplexed transmission of the data. The fault is modeled in the output links of all the CNs that use this faulted portion of the PBH bus. For example, if a PBH region has 4 levels, there would be 16 PNs in the PBH region. If a fault occurs on level 2, 4 of the PNs use the faulted portion of the bus and have their output links faulted.

PBH RECEIVER BUS

The deconcentrator network or RECEIVER BUS, sends a message to all the PNs in the PBH region using a binary tree structure. The data/address is multiplexed as in the PBH DATA/ADDRESS BUS. Control lines handshake the data between the nodes.

Faults, as with the other bus structures discussed here, can be opens or shorts which are modeled as stuck high or low values. Both the address and data fields will be modified by the fault. The level in the broadcast bus tree will determine how many PNs are effected by the fault, as in the PBH TRANSMITTER BUS. The faults will be modeled in the input links for the receiving PNs.

PBH DATA/ADDR/CNTL BUFFERS

The PBH Buffers drive the signals from the PN onto the PBH communication network. Bidirectional buffers are used for the data/address interface, and control signals handshake the data/address transfers. The output buffers drive the signal onto the PBH Transmitter Bus and the input buffers receive signals from the PBH Receiver Bus.

Any of the PBH buffers may be defective, but Fltsim only models faults in the output buffers for simplicity. Faults in the buffers are modeled as stuck high or stuck low faults. The message is multiplexed for the PBH network, so a message is transmitted in several subwords. A faulty buffer would cause several bits in the message to be faulty, so that both the address and data fields are modified by a faulty buffer. Since all the CNs in the PN that use the PBH network use a common set of buffers, if any of the buffers are faulty, all the CNs in the PN will be faulted. If an output buffer for any of the data/address signals is faulted, the output of the CN function and the message address will be modified. Thus, the faulted data will be sent using a faulty address. If any of the control lines are faulted, all transfers of data will be impaired.

PBH DEMULTIPLEXER

The PBH DEMULTIPLEXER controls the interface to the PBH bus. It controls both the sending and receiving of messages on the PBH bus. The PBH DEMUX sends the incoming message's address information to the ADDRESS DECODER and the data information to the CN MEMORY section. It will also combine the address and data information from the ADC section in order to broadcast to other CNs over the PBH network.

Faults in the PBH DEMUX will corrupt the PBH messages. Potentially, faults in the PBH DEMUX could cause single bits faults in the messages, or all the bits in the message may be faulted. Faults can also cause the whole section to malfunction, inhibiting all PBH communication for the PN. The individual stuck bits are accounted for in other hardware sections. Therefore, faults in the PBH DEMUX will disable all the PBH transfers for the entire PN. All PBH messages, both inputs and outputs, will be disabled.

Figure D1 shows how defects in the various hardware blocks are represented. A defect will occur at a location within the hardware block, as shown by the first two columns. From the area and location, specific n-graph areas and fields are identified to model the fault. If the target value to fault is not to be updated with new values, the NO_CHG column contains a Y. If updating the value is dependent on other factors, a P is indicated for a Potential NO CHANGE. Otherwise, the N indicates the value will be updated. Values that are updated will use the Stuck-At model to model the defects. The N/Y/P is similar for the ALL column. If all the CNs in a PN are to be faulted, a Y is shown. If faulting all CNs depends on other factors, a P is indicated. And if only the one CN is to be faulted, an N is indicated. The Field column indicates the target message field to fault. Either the address or data fields of the target value can be faulted.

Hardware Area	Location	N-graph	NO_CHG	ALL	Field
CN MEMORY	Link	In Link	N	N	DATA
ADDRESS DECODER	Link	Out Link	N	N	ADDR
WEIGHT	Link	Weight	N	N	DATA
LSM	LSM	Weight	P	P	DATA
DAC	Site	In Site	N	N	DATA
ADC and MUX	CN	CN	N	N	DATA
PN CONTROL	NA	Out Link	Y	Y	ADDR
PTP DATA BUS	Side	PTP Link	N	N	ADDR/DATA
PTP CNTL BUS	Side	PTP Link	Y	N	DATA
PTP CNTL/DEMUX	NA	PTP Link	Y	Y	DATA
PBH RECEIVER BUS	Level/Region	PBH In Link	N	N	ADDR/DATA
PBH TRANSMITTER BUS	Level/Region	PBH Out Link	N	N	ADDR/DATA
PBH CNTL BUS	Level/Region	PBH Out Link	Y	N	ADDR/DATA
PBH DEMULTIPLEXER	Level/Region	PBH Link	Y	N	DATA

Figure D1 - Faulted Hardware to Fault Representation.

Field	N-graph	ALL	Fault:
DATA	In Link	N	Input link
ADDR	Out Link	N	Output (Source) Address
DATA	Weight	N	Weights for input link or updated by LSM
DATA	Weight	Y	All PN Weights
DATA	In Site	N	Input Site function output
DATA	CN	N	CN function output
ADDR/DATA	PTP Link	N	Output msgs routed through faulty PTP link side
DATA	PTP Link	Y	Output msgs routed through faulty PN
ADDR/DATA	PBH In Link	N	Input msgs routed through faulty level
ADDR/DATA	PBH Out Link	N	Output msgs routed through faulty level
DATA	PBH Link	N	Input and Output msgs routed through faulty level

Figure D2 - Fault Representation to Fault Action.

Figure D2 lists the faults represented in the fault simulator and which n-graph area will model the fault. For example, a data word fault in the n-graph WEIGHT section not having the ALL bit set will be modeled in all weights specified by the link offset or a range of faults specified by the LSM. If the ALL bit is set, all the weights for the PN will be modified. Figure D2 also points out whether the input link to a CN or the output link from a CN will model fault associated in the CN links.

5. FAULT SIMULATOR DETAILED DESCRIPTION

This section will describe the detailed operation of the fault simulator program, Fltsim, used to determine the fault tolerance of the neural network emulation architecture developed at OGC. The purpose of this section is for someone maintaining or modifying Fltsim. First, a discussion of the environment that the program is run in and how to compile the program will be addressed, followed by the file naming convention used, data structures used and then a description of each routine. In this section, italicized words represent variables used in the program and capitalized words represent constants or enumerated data values. The names of subroutines used in Fltsim is shown by the routine name followed with "()". Single dimensional data arrays are shown with a "[]" following the array name, and "[][]" is used for two dimensional arrays"

Fltsim is written in C and uses the standard C libraries, which is portable to computers with the C language facilities. A make file is used to compile and link all the source code files in the system. The file, makefile, lists the dependencies between the files and the actions to be executed if the dependent files are altered. One routine uses the Unix utility Lex, which automatically is executed if the lex source file or include files have changed.

Standard file name extensions are used to describe the contents of the files. The declarations of all the structures and constants are in the file.h files, all the extern declares are in file.ext files, and all the code is in file.c files. The files pad.c and pn.c declare all the global structures uses by the fault simulator. The pad.lex file contains the source for the Unix lex routine. The lex output should be placed in the file lex.yy.c to be called by Fltsim.

Fault.h contains the definitions for the fault index bits written to the fbif.output (faulted BIF) output file. Each defined bit in the fault index indicates to the fault routine how to model the fault,

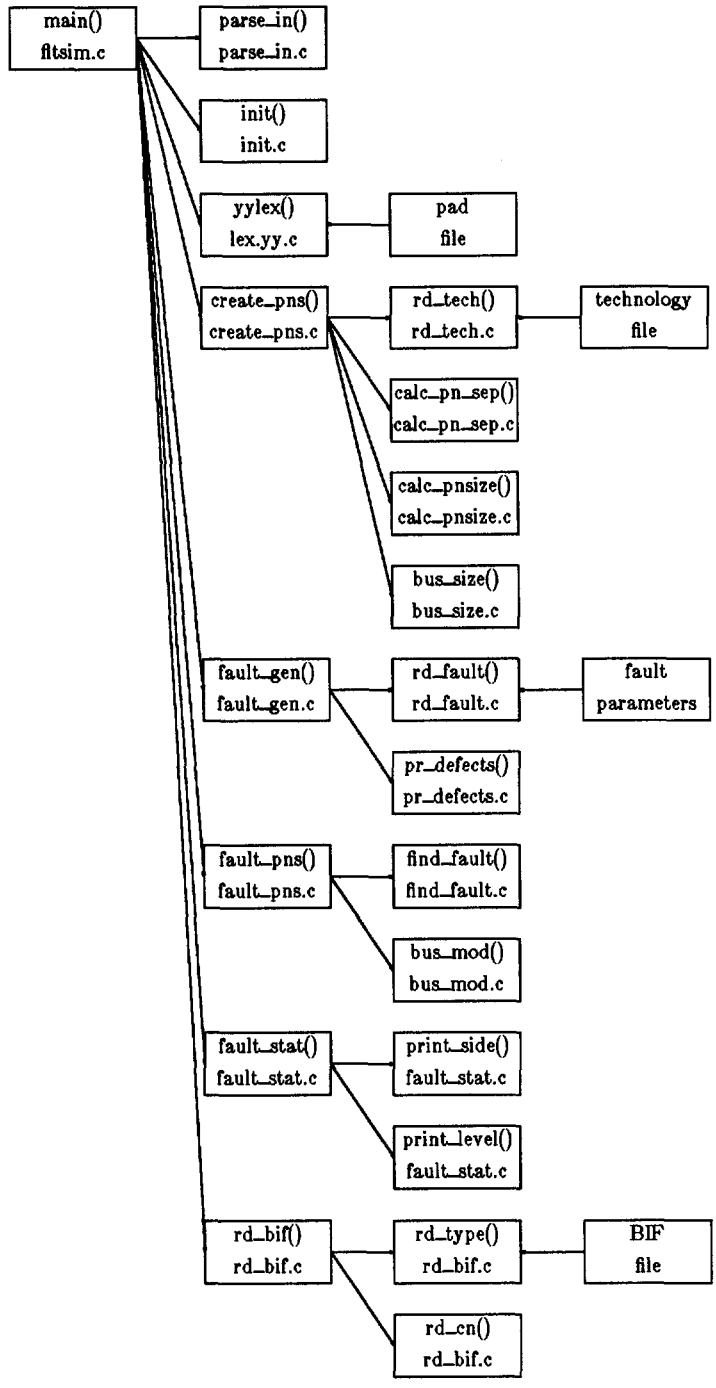


Figure E1 - Fltsim flow chart

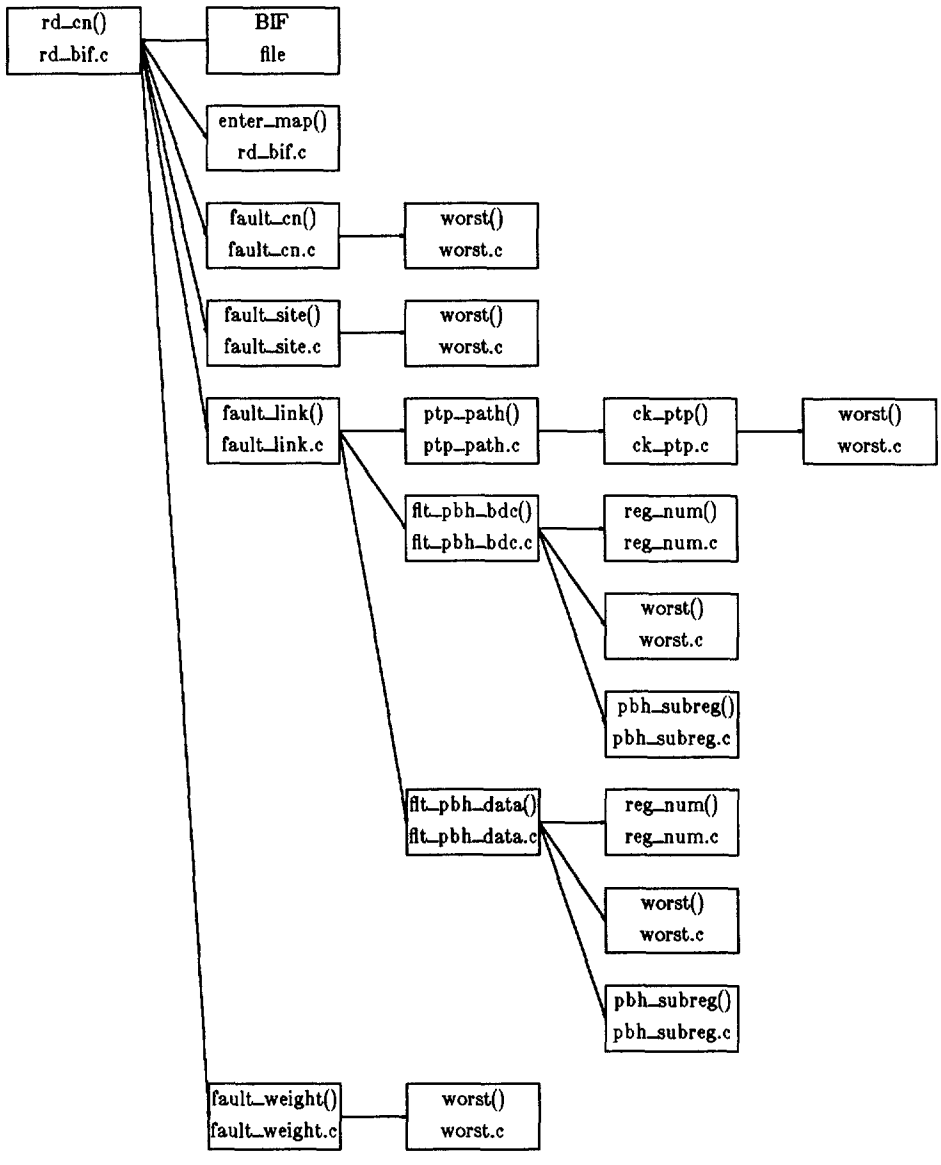


Figure E2 - Fltsim flow chart (cont)

that is, how to modify the intermediate values in the CN calculation. Fault.h will also be an include file for the user's fault routine in the architectural simulator, as discussed in the network interface appendix.

General.h contains constants non-specific to any one file or routine. The maximum number of elements for several arrays are declared here. For example, the maximum number of PNs in the simulation, the maximum number of CNs, the maximum number of faults to model, etc. These constants are later used in the structure declaration files, pad.c and pn.c, to declare array sizes containing data about each PN, CN, fault, etc.

Pn.h declares general structures and enumerated types. Pn.c references the structures defined in pn.h to declare global variables using these structures. The data structure *pn_struct* is referenced via the global variable array *pn_loc////* using an x,y PN physical coordinate. Information about each PN is stored in this structure. *Pn_areas* defines the data structure describing the physical sizes of all

the PN internal and overall dimensions, and the overall area required to implement each PN. The data structure *map* is referenced by the global variable *pn_map[]* using a PN index read from the BIF file to look up the PN's x,y coordinate. The enumerated data type, *f_types*, defines the types of faults in the fault model, namely, stuck-at-1 (*S_A_1*) and stuck-at-0 (*S_A_0*). The enumerated data type *areas* defines the hardware blocks in each PN. The n-graph is composed of several components which are listed in the enumerated type blocks. The data structure *fault_struct* is referenced via the global variable *fault[]* via a fault number. The *fault[]* variable contains information about each fault to be modeled, including the index and modifier fields to write to the fbif file, the hardware block the fault occurred in, and where the fault is modeled in the n-graph or BIF file. For some hardware blocks, such as the CN_MEMORY or WEIGHTS, the offset within the block is indicated using one of *cn_offset*, *isite_offset*, or *ilink_offset*.

When the fault parameter input file is read, the *f_params* structure is filled in by referencing *flt_parms*. The *tech_struct* data structure referenced via *tech_ptr* is filled in similarly when reading the technology file input. The data structure *bif_stats* referenced by *bif_stat* keeps track of the BIF statistics, including the number of good units (CNs, sites, links and weights) and faulted units. Additional fields will describe other pertinent statistics to be written to the statistics file.

In pn.c, there are two variables to map CN values, either from the CN index to the PN or visa versa. *Cn_map[]* uses the CN index to determine the PN number it belongs to. Given a PN number and a CN offset in the PN, the corresponding CN index can be determined using *pn_loc[x][y].cn_num[cn_offset]*. CN offsets within the hardware can be mapped to CNs and PNs in the BIF file using these data structures.

The pad.c declares global variables to describe the physical architecture. The pad variables are set when reading the pad input file. As the PAD input file is read, out of range errors are checked for, ensuring that the input data is consistent and within the ranges set in the general.h file.

The main routine contained in fltsim.c calls all the subroutines to do the various subblocks of the program. The first step is to parse the command line. The number of arguments and the character string containing the arguments are passed to parse_in(). Parse_in() opens all the specified input and output files for later reading and writing and assigns all the file pointers for each file. If no test file and/or faulted BIF file is specified, their file pointers will point to /dev/NULL to discard the data. The current date and time of execution is written to the fault statistics and test output files. All the input/output file names used by the fault simulator are written to the test output file. Parse_in(), if the -S is not specified, will generate a random seed for the random number generator is derived form the current time. Otherwise, the random number generator seed is read from the command line and will generate the same random numbers each time the program is run with the same seed value. The value of the seed is printed on the standard output and to the fault statistics file to allow any execution of Fltsim to be repeated by specifying the same seed value.

The Init() routine is next called to initialize any global arrays or variables. This routine initializes any values to their default values. CN and PN node values are set to an invalid number, -1, and the number of units in the various blocks is initialized to 0.

The physical architecture description (PAD) file is read by calling yylex(). Yylex() is the output from the lex (UNIX) lexical analysis routine and is generated from calling "lex pad.lex". The pad.lex file contains fields to search for in the PAD input file and the corresponding C code to execute for the matched fields. (Please refer to a Unix manual for details on lex.) The information in the PAD file is then read into the pad data structure (in pad.ext and pad.c). The PAD data contains the actual information required to describe the physical architecture. It includes fields such as the number of Processor Nodes (PNs), the number of Connection Nodes (CNs), how the PNs are arrayed, the connectivity of the nodes, and the number of signal lines between the nodes. See the pad description file for more details. The input values are checked for valid ranges as to not exceed array sizes declared in the constant files. If an out-of-bounds error does occur, a message is printed via stderr and program execution terminates.

Several times throughout the simulator, a test routine is called (if in TEST mode) to write intermediate Fltsim values. A pointer to the test file and a section number are passed to the test routine to determine which variables to write to the test file. For example, the test routine called with section 1 as the passed parameter will print all the parsed values produced by the lex routine. The contents of several data structures and parsed input files are included in the test output. The Test output file can be used for diagnostics or for simply keeping a complete record of the programs inputs and outputs.

With physical architecture information from the PAD file, a model of the physical structure can be built with enough detail to make the simulation accurate, but not too much detail to make it cumbersome and slow. Also, all the information to create a detailed simulation is not available. The create_pns() routine builds the model of the structure. Before the physical structure can be built, something must be known about the sizes of the devices. A technology file is read by the create_pns() routine to determine the sizes of the blocks in the architecture. Rd_tech() is passed a pointer to a technology data structure that is assigned values read from the technology file. Some of the fields in the technology file are the size of a memory cells, size of a analog-to-digital converters, size of a line buffers, line widths, digital-to-analog converter sizes, etc.

The create_pns() routine calls calc_pnsize() to calculate the size required for each PN with the technology file data. The overall PN size is the cumulation of the sizes of all the individual hardware blocks. A PN aspect ratio is specified in general.h to allow the relationship of the x,y dimensions of the PN to be specified. The sizes of these blocks are shown in figure 11. Relative locations of the individual blocks within a PN are not needed to do the simulation as the faults are placed statistically within a PN. The probability of a PN fault being assigned to a specific block is based on the relative PN block size and not the location within the block. The relative sizes of the blocks will be all that is required to place faults inside a PN. Since the relative size of the PN is small, the faults will be randomly located in the PN, and the chances of the fault being in a particular block is related to the size of the block. The fault distribution will be discussed more in the fault section.

Create_pns() calls a routine, calc_pn_sep(), to calculate the distance between the PNs. The PN separation is mainly dependent on the number of signal lines between the PNs and the size of the buffers used. The spacing between PNs is the number of data bus lines multiplied by the line width.

Create_pns() can finish creating the model since the size of PNs is known and the distance between the PNs is also known. The routine loops for each PN in the x and y dimension and assigns boundary coordinates for each PN. These boundary coordinates are used later to map faults from a physical coordinate to a specific PN.

Bus_size() is called from create_pns() to calculate the silicon area required to implement the PTP and PBH sub structures. The total area between all the PNs is calculated to provide some statistics on the fraction of area actually used for the bus signals versus free area. The number of PTP data bus lines and control lines are calculated for the entire network. Also the PBH data and control lines are calculated for the entire network. Defects in different levels of the PBH communication network will effect the operation of the network differently. So the amount of area for each level of the PBH network is calculated. The probability of a defect in a specific level is related to the amount of area required to implement that level. The bus sizes are later used by the find_fault routine to determine where the defects are located.

The next step is to determine the number of faults and the location of these faults. The fault locations are normalized, that is, they range from 0,0 to 1,1. Therefore the fault generator does not depend on the overall size of the network.

Fault parameters are read from a fault parameter file to determine the characteristics of the fault model. Rd_fault() is called to read the fault parameters into the fault data structure. The fault parameters include the defect density per square inch, the fraction of S_A_0 faults, the clustering coefficient and the ratio of the inner fault density to the outer fault density. The fault model uses a quadrat method for modeling fault clustering and a two zone radial distribution to model higher fault densities towards the wafer's edge. For the quadrat method, the wafer is divided into quadrats or

grids. The number of the inner and outer grids used by the defect generator are specified in the file general.h. The average number of defects per grid area is used to calculate the actual number of defects per grid area, as discussed in the Defect Fault Models chapter. The average number of defects per grid is the total number of defects divided by the number of grids. The overall size of the network was calculated earlier, and the average defect density is one of the fault parameters read from the fault parameter input file. Multiplying these two numbers results in the average overall number of defects. The average number of defects is divided by the number of grids to determine the average number of defects per grid area. The average defects per grid area is modified by the radial distribution factor. The two zone radial distribution allows for two zones, an inner zone and an outer zone. The ratio of the fault density for the inner and outer zones is specified in the fault parameter file, which determines the radial distribution factor. The fault density ratio will generally be less than or equal to 1. With a fault ratio of 1, the inner fault density will be identical to the outer fault density, which should be used for fault simulations of a die within a wafer. A fault ratio less than 1 indicates that the outer fault density is greater than the inner fault density, which is the normal case for a wafer. The equations below step through the calculations needed to determine the average number of faults for each grid area.

Let:

$$g_{side} = \#\ grids\ on\ a\ side\ of\ the\ outer\ zone$$

$$g_i = \#\ grids\ on\ a\ side\ of\ the\ inner\ zone,\ where\ g_{side} > g_i$$

$$f_{wi} = total\ \#\ of\ inner\ zone\ faults$$

$$f_{wo} = total\ \#\ of\ outer\ zone\ faults$$

$$f_w = average\ \#\ of\ faults\ \text{for}\ a\ wafer$$

$$r = \frac{a_i}{a_o} = inner\ outer\ fault\ density\ ratio$$

Therefore:

$$a_d = \frac{f_w}{{g_{side}}^2} = average\ \#\ faults\ per\ grid\ area$$

$$a_i = \frac{f_{wi}}{{g_i}^2} = average\ \#\ faults\ per\ inner\ grid\ area$$

$$a_o = \frac{f_{wo}}{{g_{side}}^2 - {g_i}^2} = average\ \#\ faults\ per\ outer\ grid\ area$$

Since,

$$f_w = f_{wi} + f_{wo}$$

Then,

$$f_w = a_i {g_i}^2 + a_o({g_{side}}^2 - {g_i}^2)$$

$$a_o = \frac{f_w}{g_{side}{}^2 + g_i{}^2(r-1)}$$

With the average defect density for each grid area known, the number of faults for each grid area is calculated. The number of defects in the area is determined in the pr_defects() routine, which is passed the fault parameters and the average number of faults for this area. The faults are located randomly within each quadrat using a random number generator. The type of fault (S_A_1 or S_A_0) is assigned using a random number and the ratio of S_A_1 faults to S_A_0 faults. The fault type ratio is listed in the fault parameters. A running total number of faults is used to check for overrunning any declared array sizes.

Pr_defects() uses a negative binomial distribution to determine the number of faults in quadrat. A random number with uniform distribution is mapped to the fault probability distribution. The random number is compared to the probability of 0 fault, 1 fault, 2 faults, etc. This comparison is repeated until the random number is less than the cumulative sum of the probability of x faults. The value x is returned as the number of faults. As mentioned in the Defects Fault Models chapter, the probability of x faults is:

$$Pr(x) = \frac{\Gamma(\alpha+x)}{x!\Gamma(\alpha)} \frac{(\lambda/\alpha)^x}{(1+\lambda/\alpha)^{x+\alpha}}$$

where λ is the expected number of defects in an area and α is the clustering coefficient between area on the wafers. The average number of defects in an area is λ and is read in the fault parameters file.

To calculate the gamma function, a simplification can be made shown below:

$$\frac{\Gamma(\alpha+x)}{\Gamma(\alpha)} = \alpha(\alpha+1)(\alpha+2)...(\alpha+x-1)$$

The gamma_calc() routine calculates the value of this function given α and x .

The next portion of Fltsim maps the x,y defect locations generated by fault_gen to hardware blocks in the physical architecture. Fault_pns() loops on each physical fault to map its location to a hardware block in the network. The defective operation of the hardware in the network is determined and is stored in the *fault[]* data structure defined in pn.c. A single physical fault can cause multiple faults in the n-graph. So two variables are used for the number of faults, *ph_fault_num* indicates the current physical fault number being processed and *bif_fault_num* is the current BIF or n-graph fault number which is generated from the physical faults.

Find_fault() is called to determine the faulted PN number, the faulty hardware block or area and the communication area in formation, if required, from the defect x,y coordinate. The defect x,y coordinates are normalized, so the overall die dimensions for the entire network are calculated and multiplied by the normalized x,y defect coordinate to determine the actual x,y physical fault location. The physical x,y location is used to calculate the PN x,y coordinate that is nearest to the fault. If the fault location is outside the PN boundary, then the fault is in the bus communication area and the *f_loc* variable is set to the side of the PN that the fault is located in, to be used later if the fault is in the PTP communication bus. A random number is generated to statistically derive in which bus area the fault is located. The areas for each section of bus was calculated earlier and the probability of the fault in an area is related to the size of that area. Each level of the PBH communication structure is checked for faults. If a fault is in the PBH network, the level is indicated in *f_loc*. If the fault does not occur in any of the bus signal lines, the faulted area is set to FREE which does not effect the n-graph.

If the fault is located within a PN boundary, the hardware block to be faulted is determined statistically based on the areas for each block. The *f_area* variable is set to the defective block.

The find_fault() routine returns the closest PN x,y location to the fault, the area or block the fault occurred in, and the location in the PTP or PBH communication region if required. The faulted area and PN number are stored in the *fault[]* data structure. A case statement is used to determine the effects of the fault depending on the hardware block the fault occurred in. One case statement is used for each hardware block. The case statement routines calculate the fault index, modifier, block, and any offsets (CN, site, or link) needed to model the fault in the n-graph or BIF structure. The details of each case statement are in Appendix D, Fault Effects. Fault_pns() completes the information stored in the *fault[]* data structure.

A few of the case routines place faults in multiplexed data bus lines. Multiplexing a message entails breaking it into several portions or subwords and transmitting these words serially over the data bus. The subwords are reassembled at the receiving node. A defective data bus line will cause the same bit position in each subword to be faulty. To model multiplexed data line faults, the bus_mod() routine is called, passing it the number of physical signal lines are available to transfer the data. One of the bus signal lines is chosen at random to be defective. The bit corresponding to the defective bit in the signal lines is set high in a fault modifier. The fault modifier is then shifted by the number of physical signal lines, and the defective bit position is set high again. The procedure is repeated for the entire number bits in the message, resulting in a value with high bits for all the defective bits in the transmitted message.

Fault_stat() is called by Fltsim to start writing the fault statistics to the fault statistics output file. The information contained in the *fault[]* data structure is listed in readable format. The fault index, type of fault, fault location, etc. are listed. Fault_stat() only lists information about single faults that are modeled in the hardware. In actual operation of the network, faults interact with one another. For example, a message sent between PN may have several defects along the message path, causing multiple defective bits. Fault interaction is determined when the BIF file is read, and the faults are mapped onto the n-graph. The fault interaction statistics are appended to the file in the rd_bif() routine.

Rd_bif() is next called to read the BIF input file and generate the faults in the n-graph. To place some faults at the current n-graph node being described in the BIF file, information about related nodes required to place the fault may not have been read yet. To alleviate this problem, the BIF file is read twice, the first time to enter BIF parameters into certain data structures for later use, and a second time to generate the fault indexes and modifiers for the architectural simulators, that are written into the fbif file. Different routines are called on each pass through the BIF file.

The routines get_int(), get_uint(), get_ushort(), get_short() and get_string() read and return a value from a file pointed to by the passed file pointer. The data type for the value depends on which routine is called, i.e., get_uint() reads and returns an unsigned integer. If the read fails for any reason (although it is usually due to an invalid data type) the error message string passed to the routine is printed to stderr and execution of the program stops.

Each value that is read from the BIF file can be written to the test output file with the field name or section name associated with it. If any inconsistencies or parsing errors occur, the field names aid in pinpointing the problem. Currently the BIF file contents are not written to the test file due to the potentially large size of the BIF file. In the beginning of rd_bif(), the test file pointer, *test_fp*, is set to /dev/null, discarding the BIF information. Towards the end of rd_bif(), the *test_fp* is set back to its original value. A flag can be added if needed to conditionally redirect the rd_bif() test file output.

In the beginning of each pass of reading the BIF file, several variables are initialized. The number of CNs, sites, and links in each PN are reset to 0. All the BIF fields are read from the BIF file and written to the test file on both passes. Rd_bif() reads the first line from the BIF file into a string. The string is compared to the section headings for the BIF type section and the CN section. If neither of these two sections headings are matched, an error occurs. On a match, the variable *bif_section* is set to the current BIF section being read. An entry from that BIF section is read each

time rd_type() or rd_cn() is called. When the end of the section is reached, that is, the end group or end CN keyword is read, the *bif_section* is set to NONE, and the routine returns. Rd_bif() reads another line from the BIF file for another start type or start CN keyword section header. Rd_bif() looks for section headers until the EOF, end of file, is reached. On the end of the first pass, the file pointer is rewound back to the start of the file for the next pass, where the reading procedure is executed the same as before.

The group information in the BIF file is not effected by the defects modeled in the fault simulator. Rd_type() therefore just passes information from the BIF file to the test file on each pass.

The rd_cn() routine reads the CN section in the BIF file. The BIF yields are read into local variables using one of the get_num() routines. The *procid* BIF field is used to look up the PN x,y location in the network originally specified in the PAD file. The *cn_offset* is set to the current number of CNs in this PN, which is kept track of in the *pn_loc[x][y].num_cns* data structure entry. The *num_cns* entry is incremented to indicate the current number of CNs in this PN. Note that the order of the CNs in the hardware is indicated by *cn_offset* and is determined by the order listed in the BIF file. At the level of this simulation, the ordering of CN entries within a PN does not effect the results of the simulation, although it will aid in understanding how the CN offset in the hardware is mapped to the *cn_offset* in the BIF file or n-graph. The same offset scheme is used for the sites and links. The order of the sites and links is determined by the order listed in the BIF file.

On the first pass through the BIF file, enter_map() is called to make reference tables for the second pass. Two data structures contain the reference information, *cn_map[]* and *pn_loc[x][y].cnx[ilink_offset]*. The *cn_map[]* array maps the CN index number to a PN number. The CN index read from the BIF file can be used to index the PN that contains this CN. An example of its use can be found in the BIF link section destination CN index (cnx) field. To determine the destination PN number and x,y location in the network, the *cn_map[]* array is used. *Pn_loc[x][y].cnx[]* contains all the CN indexes found in the PN with x,y location in the network and the input link offset in the hardware. The array contents are printed in the test output file.

During the second reading of the BIF file, the fault indexes and modifier values are written to the fbif file. The fbif file is a section of C code that initializes one array of structures, *flts[]* and three arrays of integers, *fcn_ptr[]*, *fsite_ptr[][]*and *flink_num[][]*. The fbif file is included with the architecture fault routines, as discussed in Appendix C. The fault indexes and modifier values in the *flts[]* array is written on the second pass through the BIF file by the fault routines fault_cn(), fault_site(), fault_link() and fault_weight(). These fault routines are described later.

Rd_sites() is called to read the site sections in the BIF file. Each site for a CN is read until the LAST bit is set in the iotype field. The variable, *isite_offset*, as discussed before, is the current input site offset for the current PN. Only input sites are considered because they have entries in the CN_MEMORY and WEIGHT hardware sections for incoming messages. The output sites send messages using the PTP and PBH hardware blocks, and these are not considered here. If reading the BIF the second time, fault_site() is called to calculate and write the site fault fields to the fbif file. Fault_site() is described later.

Rd_links() is called next to read the link section of the BIF file. The links section lists all the links for the site. Each link is read until the LAST bit is set in the lnkvec field. Again, only the input links are considered for the link_offset. Only the input link has entries in the CN_MEMORY and WEIGHTS sections. On the first pass, another reference table is set up, *pn_loc[x][y].cn_num[]*. Using the input link offset, *link_offset*, *pn_loc[x][y].cn_num[ilink_offset]* will contain the source CN index to be used in the fault_link() routine. On the second pass, fault_link() and fault_weight() are called to determine and write the fault fields for the link and weights respectively.

The routines fault_cn(), fault_site(), fault_link() and fault_weight() each search the array physical faults for any faults that will effect their corresponding BIF n-graph section. If no faults are found, a NO FAULT index and modifier consisting of zeros is written. For example, fault_cn() searches the fault array for a physical defect that will effect the current CN being processed in the BIF file. The fault array contains fields set by the fault_pns() routine, which map each defect location to

an n-graph block. Each faulted n-graph block is examined for its effect on the current BIF block. For each fault found to effect the current BIF block, a comparison is performed by the worst() routine between the previous worst case fault and the current fault. The current worst fault is initialized to a NO FAULT condition. If the two faults can be combined, the new worst fault is the combination of the two faults. Otherwise, one of the two faults is chosen as the new worst case fault.

Any two faults with stuck data bits can be combined. If both the faults have stuck-at-0 (stuck-at-1) faults, the fault modifiers are AND'ed (OR'ed) together to fault all the bits from each modifier. If one modifier has S_A_1 and the S_A_0 bits, the S_A_0 fault is arbitrarily chosen to be the dominant fault type. The modifier bits of the nondominant fault are complemented and AND'ed with the first fault's modifier bits. The rest of the worst fault cases require choosing one of the two faults as worst. If either fault is a "NO FAULT", then the other fault is chosen. If either of the two faults is a "NO CHANGE" fault, where the n-graph node of function does not get updated with new values, it is selected as the worst case fault. The worst fault is returned in the *output_flt* variable.

The fault index and modifier is written to the test file and to the faulted BIF file. The BIF statistics are updated at the end of the routine. The total number of CNs is incremented and if the fault index indicated a fault, the number of faulted CNs is incremented.

Fault_site() examines the fault array for faults effecting the current CN site. The sites are broken down into input sites and output sites. All faults are combined into a worst case fault as before. The site fault index and modifier is output to the test file and to the faulted BIF output file. The BIF statistics are updated at the end.

Fault_link() checks for faults in the message transfer between CNs. If the link is an output link, a check for address faults in the destination PN is done by flt_addr(). This routine does not check the message routing from the source PN to the destination PN, as this routing is done later. The destination PN x,y coordinate is calculated from the *pn_map[]* and *cn_map[]* arrays set on the first reading of the BIF file. *Cn_map[]* determines the destination PN number and *pn_map* determines the x and y coordinate from the PN number. The worst fault is returned in *output_flt*.

If an address fault was detected in flt_addr(), no more fault checking occurs since an address fault is the worst case fault. If no address faults were detected in the destination PN, the communication buses are checked for faults. The type of communication bus used for this link is indicated in the BIF file. If the link uses the PTP bus, the message path is determined from the source and destination PN locations and the entire route is checked for faults. The ptp_path() routine determines the route and does the fault checking. The route was arbitrarily chosen and is: move in the x dimension until the proper column is found, then move in the y dimension until the destination PN is reached. For all the PNs in the route, except for the first and last, the message enters one side and exits on another. The side is checked for faults by the ck_ptp() routine by searching through all the faults for any located in the PTP links for that side. The resultant fault fields are the worst case fault condition of the entire route.

Once the message reaches it final destination PN/CN, it is stored in the MEMORY section for the corresponding link. Any faults in the MEMORY will act identically as bus faults. So faults in the MEMORY section are searched for and combined with the PTP bus faults.

If the link uses the PBH communication bus, either flt_pbh_bdc() or flt_pbh_data() is called to calculate the link fault fields. Flt_pbh_bdc() is called for input links and flt_pbh_data() is called for output links. Faults in the PBH bus act differently depending on if they are in the signal lines in the transmitter or receiver buses, as described in the paper. These two fault routines take this difference into account.

Flt_pbh_bdc() first gets the PBH region in which the current CN is located. Reg_num() returns the PBH region that the PN is listed in and where it is located within the PBH tree. The location is indicated by an offset in the list of PNs in the region. Then all the faults are searched for any in the PBH input links. The region that have PBH input link faults is determined and compared to the region that the current CN is in. If they are in the same PBH region, they are checked to see if

the fault location in the PBH communication tree effects both CNs. The level in the PBH tree that the fault occurs is indicated by the *flt_ptr->loc* variable. All nodes that are descendents of this faulted node will be faulty. The routine pbh_subreg() returns TRUE if the two CNs are in the same subregion or subtree. The worst fault is determined as before. Flt_pbh_data() works similarly as flt_pbh_bdc() except the PBH output link fault are searched for. To reiterate the reason for modeling the PBH receiver and transmitter buses separately is due to the differing actions of the faults in these two buses. A fault in the receiver bus will effect all input messages to PNs in that PBH subregion. A fault in the transmitter bus will effect all messages coming from the PNs in the faulted PBH subtree. Receiver faults effect messages to PNs and transmitter faults effect messages from PNs.

The fault_link() routine completes by writing the fault fields to the test routine and faulted BIF file and updating the BIF link statistics.

The fault_weight() routine is called next to check for any faults effecting the weights for the input links. The fault array is searched for faults effecting the current link read from the BIF file. The resultant worst case fault is written to the test file and to the faulted BIF output files. The BIF statistics for the weights are updated.

After reading the BIF file the second time and generating all the BIF fault fields, the rd_bif() routine writes index routines to access the fault fields by the fault routines. These are described in more detail in the fault simulation interface appendix.

The next step in the rd_bif() routine is to write the results of the BIF statistics to the fstat output file. The fields written here quickly summarize the effects of faults on the network. Percentages of faulted CNs, sites, links, and weights are shown, with the addition of more fields as required later.

Fltsim completes its operation by closing all the opened files. The Fault Simulator execution is completed.