1. Intro. This program inputs a gate graph and finds a set of test vectors that will detect most of the "single stuck-at faults" for those gates. It outputs the names of the faults that it wasn't able to cover.

The first command-line parameter is the name of the graph, in standard Stanford GraphBase format as defined in GB_GATES. The second parameter is a seed for the random number generator. An optional third parameter is the probability that an input bit is set to 1. (It's a floating point number between 0 and 1. It's 0.5 by default.) An optional fourth parameter is the name of an output file for the test patterns. An optional fifth parameter is the name of an output file for the lists of previously undetected faults that are detected by each pattern.

If the graph has n vertices (gates), there are $2n$ stuck-at faults. The two faults for a gate named `gg` are called `0gg` and `1gg`, representing the hypothesis that a faulty `gg` always produces the value 0 or 1, respectively, irrespective of its inputs.

(Important note: I wrote the above before realizing that I should really apply these methods to "wires," not "gates." One should use this program with a graph output by GATES-TO-WIRES.)

The method is simply to try random inputs and to see what new faults they detect, until finding no more. A given sequence of inputs is tested by computing its behavior with respect to all not-yet-covered faults, using bitwise operations to handle 64 cases at once.

If the given seed value is k times 1000, or more, we will continue we've done at least $k + 1$ consecutive passes over the circuit without finding a new pattern.

```
#include "gb_graph.h"
#include "gb_gates.h"
#include "gb_flip.h"
#include "gb_save.h"
  unsigned long long **bits;
  Vertex **faults;
  int seed;
  double bias;
  unsigned long long thresh = 2147483648 ≫ 1;
  FILE *pat_file, *fault_file;

  main(int argc, char *argv[])
  {
    register int i, j, k, n, r, s;
    register unsigned long long udefault, vdefault;
    register Vertex *u, *v;
    register Arc *a;
    register Graph *g;
    int faults_left, faults_found, tolerance;

    ⟨ Process the command line 2 ⟩;
    n = g→n;
    ⟨ Allocate the auxiliary arrays 3 ⟩;
    ⟨ Prepend all latches to the list of outputs 5 ⟩;
    ⟨ Initialize the list of faults remaining 6 ⟩;
    fprintf(stderr, "(considering␣%d␣possible␣single-stuck-at␣faults)\n", faults_left);
    tolerance = seed/1000;
    while (faults_left) {
      faults_found = 0;
      ⟨ Pass over the circuit with random inputs 7 ⟩;
      faults_left −= faults_found;
      fprintf(stderr, "(found␣%d;␣%d␣left)\n", faults_found, faults_left);
      if (faults_found) {
        tolerance = seed/1000;
        if (pat_file) ⟨ Output the current test pattern 17 ⟩;
      } else if (−−tolerance < 0) break;
    }
    ⟨Print out the remaining faults 16⟩;
  }
```

2. ⟨Process the command line 2⟩ ≡

```
  if (argc < 3 ∨ argc > 6 ∨ sscanf(argv[2], "%d", &seed) ≠ 1) {
    fprintf(stderr, "Usage:␣%s␣gates.gb␣seed␣[bias]␣[patternfile]␣[faultfile]\n", argv[0]);
    exit(−1);
  }
  g = restore_graph(argv[1]);
  if (¬g) {
    fprintf(stderr, "I␣can't␣restore␣the␣graph␣`%s'!\n", argv[1]);
    exit(−2);
  }
  if (argc > 3) {
    if (sscanf(argv[3], "%lf", &bias) ≠ 1 ∨ bias ≤ 0.0 ∨ bias ≥ 1.0) {
      fprintf(stderr, "The␣bias␣should␣be␣strictly␣between␣0.0␣and␣1.0␣(default␣0.5)!\n");
      exit(−8);
    }
    thresh = bias ∗ 2147483648.0;      /∗ 2^31 ∗/
    if (argc > 4) {
      pat_file = fopen(argv[4], "w");
      if (¬pat_file) {
        fprintf(stderr, "I␣can't␣open␣the␣pattern␣file␣`%s'␣for␣writing!\n", argv[4]);
        exit(−3);
      }
      if (argc > 5) {
        fault_file = fopen(argv[5], "w");
        if (¬fault_file) {
          fprintf(stderr, "I␣can't␣open␣the␣fault␣file␣`%s'␣for␣writing!\n", argv[5]);
          exit(−4);
        }
      }
    }
  }
  gb_init_rand(seed);
```

This code is used in section 1.

3. ⟨ Allocate the auxiliary arrays 3 ⟩ ≡

```
bits = (unsigned long long **) malloc((n + 1) * sizeof(unsigned long long *));
if (¬bits) {
  fprintf(stderr, "I␣can't␣allocate␣the␣bits␣array!\n");
  exit(−5);
}
k = 1 + (n ≫ 5);   /* this many octabytes needed for the first round of simulation */
if (sizeof(unsigned long long) ≠ 8) {
  fprintf(stderr, "Sorry,␣I␣wrote␣this␣code␣assuming␣64-bit␣words!\n");
  exit(−6);
}
for (j = 0; j < n; j++) {
  bits[j] = (unsigned long long *) malloc(k * sizeof(unsigned long long));
  if (¬bits[j]) {
    fprintf(stderr, "I␣can't␣allocate␣the␣array␣bits[%d]!\n", j);
    exit(−7);
  }
}
```

See also section 4.

This code is used in section 1.

4. The kth fault that is still untested is identified in *faults*[k].

Some C hacking is used to represent stuck-at faults as pointers to vertices: If v points to a gate, we add 1 to the numerical value of v to indicate "v stuck at 1."

```
#define stuck_at_one(v) (Vertex *)((unsigned long long)(v) + 1)
#define how_stuck(v) (int)((unsigned long long)(v) & 1)
#define clean(v) ((Vertex *)((unsigned long long)(v) & −2))
```

⟨ Allocate the auxiliary arrays 3 ⟩ +≡

```
faults = (Vertex **) malloc((n + n + 1) * sizeof(Vertex *));
if (¬faults) {
  fprintf(stderr, "I␣can't␣allocate␣the␣faults␣array!\n");
  exit(−8);
}
```

5. If g contains latches, they must follow the inputs and precede all other gates. I'll want to treat the source of each latch as an output, in a "combinational" circuit that computes a function of inputs and latches.

In this program the *val* field of vertex v is nonzero if and only if v is an output.

⟨ Prepend all latches to the list of outputs 5 ⟩ ≡

```
for (v = g→vertices; v < g→vertices + n ∧ v→typ ≡ 'I'; v++) v→val = 0;
for ( ; v < g→vertices + n ∧ v→typ ≡ 'L'; v++) {
  v→val = 0;
  a = gb_virgin_arc( );
  a→next = g→outs;
  a→tip = v→alt;
  g→outs = a;
}
for ( ; v < g→vertices + n; v++) v→val = 0;
for (k = 0, a = g→outs; a; a = a→next) a→tip→val = ++k;
```

This code is used in section 1.

6. A fault at gate v is unresolved if and only if *v→stuck0* or *v→stuck1* is nonzero.

```
#define stuck0 u.I    /* utility field u of a vertex */
#define stuck1 v.I    /* utility field v of a vertex */
```

⟨ Initialize the list of faults remaining 6 ⟩ ≡

```
  faults_left = 0;
  for (v = g→vertices; v < g→vertices + n; v++) v→stuck0 = v→stuck1 = 1, faults_left += 2;
```

This code is used in section 1.

7. Passing over the circuit. The heart of this computation is a loop in which we pass over the gates one by one, evaluating them with respect to each of the pending fault scenarios. Variable s is the current number of faults under consideration.

We pack 64 scenarios per octabyte in the *bits* table of each vertex; fault k is bit $k\,\&\,{}^\#\texttt{3f}$ from the right in $bits[j][k \gg 6]$, when $v = g\to vertices + j$.

"Fault 0" is the normal case where everything is operating correctly. This default value for a gate, which appears in the least significant bit of $bits[j][0]$, is assumed to apply in all scenarios $> s$. (Hey, "default" means "no faults," get it?)

During this processing we set *vdefault* to 0 or -1, according as the default value for v is 0 or 1.

The value of s at vertex v is stored in $v\to size$.

```
#define size w.I    /* utility field w of a vertex */
⟨ Pass over the circuit with random inputs 7 ⟩ ≡
  for (s = j = 0; j < n; j++) {
    v = g→vertices + j;
    ⟨ Compute vdefault and bits[j] 8 ⟩;
    if (v→stuck0) {
      s++;
      faults[s] = v;
      if ((s & #3f) ≡ 0) bits[j][s ≫ 6] = vdefault;
      bits[j][s ≫ 6] &= ∼(1ULL ≪ (s & #3f));
    }
    if (v→stuck1) {
      s++;
      faults[s] = stuck_at_one(v);
      if ((s & #3f) ≡ 0) bits[j][s ≫ 6] = vdefault;
      bits[j][s ≫ 6] |= 1ULL ≪ (s & #3f);
    }
    v→size = s;
    if (v→val) ⟨ See if we've covered any faults 15 ⟩;
  }
```

This code is used in section 1.

```
8. ⟨ Compute vdefault and bits[j] 8 ⟩ ≡
  switch (v→typ) {
  case 'I': case 'L': ⟨ Assign a random input 9 ⟩; break;
  case '&': ⟨ Process an AND gate 12 ⟩; break;
  case '|': ⟨ Process an OR gate 13 ⟩; break;
  case '^': ⟨ Process an XOR gate 14 ⟩; break;
  case '~': ⟨ Process an inverter 10 ⟩; break;
  case 'F': ⟨ Process a clone gate 11 ⟩; break;
  default:
    fprintf(stderr, "Vertex␣%s␣(%d)␣has␣unknown␣gate␣type␣'%c'!\n", v→name, j, (char) v→typ);
    exit(−10);
  }
```

This code is used in section 7.

```
9. ⟨ Assign a random input 9 ⟩ ≡
  vdefault = −(gb_next_rand() < thresh);
  for (k = 0; k ≤ s ≫ 6; k++) bits[j][k] = vdefault;
```

This code is used in section 8.

10. ⟨ Process an inverter 10 ⟩ ≡

```
if (¬v→arcs ∨ v→arcs→next) {
  fprintf(stderr, "Inverter␣%s␣(%d)␣should␣have␣exactly␣one␣operand!\n", v→name, j);
  exit(−11);
}
u = v→arcs→tip;
i = u − g→vertices;
udefault = −(bits[i][0] & 1), r = u→size;
vdefault = ∼udefault;
for (k = 0; k ≤ r ≫ 6; k++) bits[j][k] = ∼bits[i][k];
for ( ; k ≤ s ≫ 6; k++) bits[j][k] = vdefault;
```

This code is used in section 8.

11. A "clone gate" is really a wire that's a fanout branch. It should copy the value of its lone parameter (which is a fanout stem).

⟨ Process a clone gate 11 ⟩ ≡

```
if (¬v→arcs ∨ v→arcs→next) {
  fprintf(stderr, "Fanout␣branch␣%s␣(%d)␣should␣have␣exactly␣one␣operand!\n", v→name, j);
  exit(−11);
}
u = v→arcs→tip;
i = u − g→vertices;
udefault = −(bits[i][0] & 1), r = u→size;
vdefault = udefault;
for (k = 0; k ≤ r ≫ 6; k++) bits[j][k] = bits[i][k];
for ( ; k ≤ s ≫ 6; k++) bits[j][k] = vdefault;
```

This code is used in section 8.

12. I think some interesting optimization is possible here (and elsewhere in this program), but I'm eschewing it today.

⟨ Process an AND gate 12 ⟩ ≡

```
vdefault = −1;
for (k = 0; k ≤ s ≫ 6; k++) bits[j][k] = vdefault;
for (a = v→arcs; a; a = a→next) {
  u = a→tip, i = u − g→vertices;
  udefault = −(bits[i][0] & 1), r = u→size;
  for (k = 0; k ≤ r ≫ 6; k++) bits[j][k] &= bits[i][k];
  if (udefault ≡ 0) {
    vdefault = 0;
    for ( ; k ≤ s ≫ 6; k++) bits[j][k] = 0;
  }
}
```

This code is used in section 8.

13. ⟨ Process an OR gate 13 ⟩ ≡

```
$vdefault = 0;$
**for** $(k = 0;\ k \le s \gg 6;\ k{+}{+})\ bits[j][k] = vdefault;$
**for** $(a = v\text{→}arcs;\ a;\ a = a\text{→}next)$ {
  $u = a\text{→}tip, i = u - g\text{→}vertices;$
  $udefault = -(bits[i][0] \mathbin{\&} 1), r = u\text{→}size;$
  **for** $(k = 0;\ k \le r \gg 6;\ k{+}{+})\ bits[j][k] \mathrel{|}= bits[i][k];$
  **if** $(udefault)$ {
    $vdefault = -1;$
    **for** $(\ ;\ k \le s \gg 6;\ k{+}{+})\ bits[j][k] = -1;$
  }
}
```

This code is used in section 8.

14. ⟨ Process an XOR gate 14 ⟩ ≡

```
$vdefault = 0;$
**for** $(k = 0;\ k \le s \gg 6;\ k{+}{+})\ bits[j][k] = vdefault;$
**for** $(a = v\text{→}arcs;\ a;\ a = a\text{→}next)$ {
  $u = a\text{→}tip, i = u - g\text{→}vertices;$
  $udefault = -(bits[i][0] \mathbin{\&} 1), r = u\text{→}size;$
  **for** $(k = 0;\ k \le r \gg 6;\ k{+}{+})\ bits[j][k] \oplus= bits[i][k];$
  **if** $(udefault)$ {
    $vdefault \oplus= udefault;$
    **for** $(\ ;\ k \le s \gg 6;\ k{+}{+})\ bits[j][k] \oplus= -1;$
  }
}
```

This code is used in section 8.

15. When we reach an output gate, any scenarios that don't agree with *vdefault* are now covered by the current random inputs.

```
⟨ See if we've covered any faults 15 ⟩ ≡
  {
    for ($k = 0$; $k \le s \gg 6$; $k{+}{+}$)
      if ($bits[j][k] \oplus vdefault$) {
        $udefault = bits[j][k] \oplus vdefault$;
        for ($i = 0$; $i < 64$; $i{+}{+}$)
          if ($udefault$ & $(1_{\rm ULL} \ll i)$) {
            $u = faults[(k \ll 6) + i]$;
            if ($how\_stuck(u)$) {
              $u = clean(u)$;
              if ($u\to stuck1$) {
                $faults\_found{+}{+}, u\to stuck1 = 0$;
                if ($fault\_file$) $fprintf(fault\_file,$ "␣1%s", $u\to name)$;
              }
            } else {
              if ($u\to stuck0$) {
                $faults\_found{+}{+}, u\to stuck0 = 0$;
                if ($fault\_file$) $fprintf(fault\_file,$ "␣0%s", $u\to name)$;
              }
            }
          }
      }
  }
```

This code is used in section 7.

16. Output. Now that the algorithm is fully implemented, we need only write the code that communicates its results.

```
⟨ Print out the remaining faults 16 ⟩ ≡
  for (k = 1; k ≤ faults_left; k++) {
    v = clean(faults[k]);
    printf("␣%c%s", how_stuck(faults[k]) ? '1' : '0', v→name);
  }
  printf("\n");
  if (pat_file) {
    fprintf(stderr, "Test␣patterns␣written␣on␣file␣`%s'", argv[4]);
    if (fault_file) fprintf(stderr, ";␣covered␣faults␣written␣on␣file␣`%s'.\n", argv[5]);
    else fprintf(stderr, ".\n");
  }
```

This code is used in section 1.

17. Each test pattern is a string of input bits (`0` or `1`), followed by `->`, followed by a string of correct output bits, followed by a newline character.

```
⟨ Output the current test pattern 17 ⟩ ≡
  {
    for (j = 0; j < n; j++) {
      v = g→vertices + j;
      if (v→typ ≡ 'I' ∨ v→typ ≡ 'L') fprintf(pat_file, "%c", (char)('0' + (bits[j][0] & 1)));
    }
    fprintf(pat_file, "->");
    for (a = g→outs; a; a = a→next) {
      v = a→tip;
      fprintf(pat_file, "%c", (char)('0' + (bits[v − g→vertices][0] & 1)));
    }
    fprintf(pat_file, "\n");
    if (fault_file) fprintf(fault_file, "\n");
  }
```

This code is used in section 1.

18. Index.

a: 1.
alt: 5.
Arc: 1.
arcs: 10, 11, 12, 13, 14.
argc: 1, 2.
argv: 1, 2, 16.
bias: 1, 2.
bits: 1, 3, 7, 9, 10, 11, 12, 13, 14, 15, 17.
clean: 4, 15, 16.
exit: 2, 3, 4, 8, 10, 11.
fault_file: 1, 2, 15, 16, 17.
faults: 1, 4, 7, 15, 16.
faults_found: 1, 15.
faults_left: 1, 6, 16.
fopen: 2.
fprintf: 1, 2, 3, 4, 8, 10, 11, 15, 16, 17.
g: 1.
gb_init_rand: 2.
gb_next_rand: 9.
gb_virgin_arc: 5.
Graph: 1.
how_stuck: 4, 15, 16.
i: 1.
j: 1.
k: 1.
main: 1.
malloc: 3, 4.
n: 1.
name: 8, 10, 11, 15, 16.
next: 5, 10, 11, 12, 13, 14, 17.
outs: 5, 17.
pat_file: 1, 2, 16, 17.
printf: 16.
r: 1.
restore_graph: 2.
s: 1.
seed: 1, 2.
size: 7, 10, 11, 12, 13, 14.
sscanf: 2.
stderr: 1, 2, 3, 4, 8, 10, 11, 16.
stuck_at_one: 4, 7.
stuck0: 6, 7, 15.
stuck1: 6, 7, 15.
thresh: 1, 2, 9.
tip: 5, 10, 11, 12, 13, 14, 17.
tolerance: 1.
typ: 5, 8, 17.
u: 1.
udefault: 1, 10, 11, 12, 13, 14, 15.
v: 1.
val: 5, 7.
vdefault: 1, 7, 9, 10, 11, 12, 13, 14, 15.
Vertex: 1, 4.
vertices: 5, 6, 7, 10, 11, 12, 13, 14, 17.

⟨ Allocate the auxiliary arrays 3, 4 ⟩ Used in section 1.
⟨ Assign a random input 9 ⟩ Used in section 8.
⟨ Compute *vdefault* and *bits*$[j]$ 8 ⟩ Used in section 7.
⟨ Initialize the list of faults remaining 6 ⟩ Used in section 1.
⟨ Output the current test pattern 17 ⟩ Used in section 1.
⟨ Pass over the circuit with random inputs 7 ⟩ Used in section 1.
⟨ Prepend all latches to the list of outputs 5 ⟩ Used in section 1.
⟨ Print out the remaining faults 16 ⟩ Used in section 1.
⟨ Process a clone gate 11 ⟩ Used in section 8.
⟨ Process an AND gate 12 ⟩ Used in section 8.
⟨ Process an OR gate 13 ⟩ Used in section 8.
⟨ Process an XOR gate 14 ⟩ Used in section 8.
⟨ Process an inverter 10 ⟩ Used in section 8.
⟨ Process the command line 2 ⟩ Used in section 1.
⟨ See if we've covered any faults 15 ⟩ Used in section 7.

GATES-STUCK

	Section	Page
Intro	1	1
Passing over the circuit	7	5
Output	16	9
Index	18	10