Important: Before reading MULTIPLY, please read or at least skim the program for GB_GATES.

1. Introduction. This demonstration program uses graphs constructed by the *prod* procedure in the GB_GATES module to produce an interactive program called `multiply`, which multiplies and divides small numbers the slow way—by simulating the behavior of a logical circuit, one gate at a time.

The program assumes that UNIX conventions are being used. Some code in sections listed under 'UNIX dependencies' in the index might need to change if this program is ported to other operating systems.

To run the program under UNIX, say '`multiply` m n [*seed*]', where m and n are the sizes of the numbers to be multiplied, in bits, and where *seed* is given if and only if you want the multiplier to be a special-purpose circuit for multiplying a given m-bit number by a randomly chosen n-bit constant.

The program will prompt you for two numbers (or for just one, if the random constant option has been selected), and it will use the gate network to compute their product. Then it will ask for more input, and so on.

2. Here is the general layout of this program, as seen by the C compiler:

```
#include "gb_graph.h"     /* the standard GraphBase data structures */
#include "gb_gates.h"     /* routines for gate graphs */
  ⟨ Preprocessor definitions ⟩
  ⟨ Global variables 4 ⟩
  ⟨ Handy subroutines 10 ⟩
  main(argc, argv)
      int argc;     /* the number of command-line arguments */
      char *argv[];     /* an array of strings containing those arguments */
  {
    ⟨ Declare variables that ought to be in registers 5 ⟩;
    ⟨ Obtain m, n, and optional seed from the command line 6 ⟩;
    ⟨ Make sure m and n are valid; generate the prod graph g 3 ⟩;
    if (seed < 0)     /* no seed given */
      printf("Here␣I␣am,␣ready␣to␣multiply␣%ld-bit␣numbers␣by␣%ld-bit␣numbers.\n", m, n);
    else {
      g = partial_gates(g, m, 0L, seed, buffer);
      if (g) {
        ⟨ Set y to the decimal value of the second input 9 ⟩;
        printf("OK,␣I'm␣ready␣to␣multiply␣any␣%ld-bit␣number␣by␣%s.\n", m, y);
      } else {     /* there was enough memory to make the original g, but not enough to reduce it; this
            probably can't happen, but who knows? */
        printf("Sorry,␣I␣couldn't␣process␣the␣graph␣(trouble␣code␣%ld)!\n", panic_code);
        return −9;
      }
    }
    printf("(I'm␣simulating␣a␣logic␣circuit␣with␣%ld␣gates,␣depth␣%ld.)\n", g→n, depth(g));
    while (1) {
      ⟨ Prompt for one or two numbers; break if unsuccessful 7 ⟩;
      ⟨ Use the network to compute the product 11 ⟩;
      printf("%sx%s=%s%s.\n", x, y, (strlen(x) + strlen(y) > 35 ? "\n␣" : ""), z);
    }
    return 0;     /* normal exit */
  }
```

3. ⟨ Make sure m and n are valid; generate the *prod* graph g 3 ⟩ ≡

```
if (m < 2) m = 2;
if (n < 2) n = 2;
if (m > 999 ∨ n > 999) {
  printf("Sorry,␣I'm␣set␣up␣only␣for␣precision␣less␣than␣1000␣bits.\n");
  return −1;
}
if ((g = prod(m, n)) ≡ Λ) {
  printf("Sorry,␣I␣couldn't␣generate␣the␣graph␣(not␣enough␣memory␣for␣%s)!\n", panic_code ≡
      no_room ? "the␣gates" : panic_code ≡ alloc_fault ? "the␣wires" : "local␣optimization");
  return −3;
}
```

This code is used in section 2.

4. To figure the maximum length of strings x and y, we note that $2^{999} \approx 5.4 \times 10^{300}$.

⟨ Global variables 4 ⟩ ≡

```
Graph *g;   /* graph that defines a logical network for multiplication */
long m, n;   /* length of binary numbers to be multiplied */
long seed;   /* optional seed value, or −1 */
char x[302], y[302], z[603];   /* input and output numbers, as decimal strings */
char buffer[2000];   /* workspace for communication between routines */
```

This code is used in section 2.

5. ⟨ Declare variables that ought to be in registers 5 ⟩ ≡

```
register char *p, *q, *r;   /* pointers for string manipulation */
register long a, b;   /* amounts being carried over while doing radix conversion */
```

This code is used in section 2.

6. ⟨ Obtain m, n, and optional *seed* from the command line 6 ⟩ ≡

```
if (argc < 3 ∨ argc > 4 ∨ sscanf(argv[1], "%ld", &m) ≠ 1 ∨ sscanf(argv[2], "%ld", &n) ≠ 1) {
  fprintf(stderr, "Usage:␣%s␣m␣n␣[seed]\n", argv[0]);
  return −2;
}
if (m < 0) m = −m;   /* maybe the user attached '-' to the argument */
if (n < 0) n = −n;
seed = −1;
if (argc ≡ 4 ∧ sscanf(argv[3], "%ld", &seed) ≡ 1 ∧ seed < 0) seed = −seed;
```

This code is used in section 2.

7. This program may not be user-friendly, but at least it is polite.

```
#define prompt(s)
        { printf(s); fflush(stdout);      /* make sure the user sees the prompt */
          if (fgets(buffer, 999, stdin) ≡ Λ) break; }
#define retry(s, t)
        { printf(s); goto t; }
```

⟨ Prompt for one or two numbers; **break** if unsuccessful 7 ⟩ ≡

```
step1: prompt("\nNumber,␣please?␣");
  for (p = buffer; *p ≡ '0'; p++) ;      /* bypass leading zeroes */
  if (*p ≡ '\n') {
    if (p > buffer) p--;      /* zero is acceptable */
    else break;      /* empty input terminates the run */
  }
  for (q = p; *q ≥ '0' ∧ *q ≤ '9'; q++) ;      /* check for digits */
  if (*q ≠ '\n')
    retry("Excuse␣me...␣I'm␣looking␣for␣a␣nonnegative␣sequence␣of␣decimal␣digits.", step1);
  *q = 0;
  if (strlen(p) > 301) retry("Sorry,␣that's␣too␣big.", step1);
  strcpy(x, p);
  if (seed < 0) {
    ⟨ Do the same thing for y instead of x 8 ⟩;
  }
```

This code is used in section 2.

8. ⟨ Do the same thing for y instead of x 8 ⟩ ≡

```
step2: prompt("Another?␣");
  for (p = buffer; *p ≡ '0'; p++) ;      /* bypass leading zeroes */
  if (*p ≡ '\n') {
    if (p > buffer) p--;      /* zero is acceptable */
    else break;      /* empty input terminates the run */
  }
  for (q = p; *q ≥ '0' ∧ *q ≤ '9'; q++) ;      /* check for digits */
  if (*q ≠ '\n')
    retry("Excuse␣me...␣I'm␣looking␣for␣a␣nonnegative␣sequence␣of␣decimal␣digits.", step2);
  *q = 0;
  if (strlen(p) > 301) retry("Sorry,␣that's␣too␣big.", step2);
  strcpy(y, p);
```

This code is used in section 7.

9. The binary value chosen at random by *partial_gates* appears as a string of 0s and 1s in *buffer*, in little-endian order. We compute the corresponding decimal value by repeated doubling.

If the value turns out to be zero, the whole network will have collapsed. Otherwise, however, the m inputs from the first operand will all remain present, because they all affect the output.

⟨ Set y to the decimal value of the second input 9 ⟩ ≡

```
*y = '0'; *(y + 1) = 0;    /* now y is "0" */
for (r = buffer + strlen(buffer) - 1; r >= buffer; r--) {
    /* we will set y = 2y + t where t is the next bit, *r */
  if (*y >= '5') a = 0, p = y;
  else a = *y - '0', p = y + 1;
  for (q = y; *p; a = b, p++, q++) {
    if (*p >= '5') {
      b = *p - '5';
      *q = 2 * a + '1';
    } else {
      b = *p - '0';
      *q = 2 * a + '0';
    }
  }
  if (*r == '1') *q = 2 * a + '1';
  else *q = 2 * a + '0';
  *++q = 0;    /* terminate the string */
}
if (strcmp(y, "0") == 0) {
  printf("Please try another seed value; %d makes the answer zero!\n", seed);
  return (-5);
}
```

This code is used in section 2.

10. Using the network. The reader of the code in the previous section will have noticed that we are representing high-precision decimal numbers as strings. We might as well do that, since the only operations we need to perform on them are input, output, doubling, and halving. In fact, arithmetic on strings is kind of fun, if you like that sort of thing.

Here is a subroutine that converts a decimal string to a binary string. The decimal string is big-endian as usual, but the binary string is little-endian. The decimal string is decimated in the process; it should end up empty, unless the original value was too big.

```
⟨ Handy subroutines 10 ⟩ ≡
  decimal_to_binary(x, s, n)
      char *x;    /* decimal string */
      char *s;    /* binary string */
      long n;    /* length of s */
  { register long k;
    register char *p, *q;    /* pointers for string manipulation */
    register long r;    /* remainder */
    for (k = 0; k < n; k++, s++) {
      if (*x ≡ 0) *s = '0';
      else {    /* we will divide x by 2 */
        if (*x > '1') p = x, r = 0;
        else p = x + 1, r = *x − '0';
        for (q = x; *p; p++, q++) {
          r = 10 * r + *p − '0';
          *q = (r ≫ 1) + '0';
          r = r & 1;
        }
        *q = 0;    /* terminate string x */
        *s = '0' + r;
      }
    }
    *s = 0;    /* terminate the output string */
  }
```

See also section 13.

This code is used in section 2.

11. ⟨ Use the network to compute the product 11 ⟩ ≡

```
strcpy(z, x);
decimal_to_binary(z, buffer, m);
if (*z) {
  printf("(Sorry, %s has more than %ld bits.)\n", x, m);
  continue;
}
if (seed < 0) {
  strcpy(z, y);
  decimal_to_binary(z, buffer + m, n);
  if (*z) {
    printf("(Sorry, %s has more than %ld bits.)\n", y, n);
    continue;
  }
}
if (gate_eval(g, buffer, buffer) < 0) {
  printf("??? An internal error occurred!");
  return 666;    /* this can't happen */
}
⟨ Convert the binary number in buffer to the decimal string z 12 ⟩;
```

This code is used in section 2.

12. The remaining task is almost identical to what we needed to do when computing the value of y after a random seed was specified. But this time the binary number in *buffer* is big-endian.

⟨ Convert the binary number in *buffer* to the decimal string z 12 ⟩ ≡

```
*z = '0'; *(z + 1) = 0;
for (r = buffer; *r; r++) {    /* we'll set z = 2z + t where t is the next bit, *r */
  if (*z ≥ '5') a = 0, p = z;
  else a = *z - '0', p = z + 1;
  for (q = z; *p; a = b, p++, q++) {
    if (*p ≥ '5') {
      b = *p - '5';
      *q = 2 * a + '1';
    } else {
      b = *p - '0';
      *q = 2 * a + '0';
    }
  }
  if (*r ≡ '1') *q = 2 * a + '1';
  else *q = 2 * a + '0';
  *++q = 0;    /* terminate the string */
}
```

This code is used in section 11.

13. Calculating the depth. The depth of a gate network produced by GB_GATES is easily discovered by making one pass over the vertices. An input gate or a constant has depth 0; every other gate has depth one greater than the maximum of its inputs.

This routine is more general than it needs to be for the circuits output by *prod*. The result of a latch is considered to have depth 0.

Utility field $u.I$ is set to the depth of each individual gate.

```
#define dp u.I
⟨Handy subroutines 10⟩ +≡
  long depth(g)
      Graph *g;   /* graph with gates as vertices */
  { register Vertex *v;   /* the current vertex of interest */
    register Arc *a;   /* the current arc of interest */
    long d;   /* depth of current vertex */
    if (¬g) return −1;   /* no graph supplied! */
    for (v = g→vertices; v < g→vertices + g→n; v++) {
      switch (v→typ) {   /* branch on type of gate */
      case 'I': case 'L': case 'C': v→dp = 0; break;
      default: ⟨Set d to the maximum depth of an operand of v 14⟩;
        v→dp = 1 + d;
      }
    }
    ⟨Set d to the maximum depth of an output of g 15⟩;
    return d;
  }
```

14. ⟨Set d to the maximum depth of an operand of v 14⟩ ≡

```
  d = 0;
  for (a = v→arcs; a; a = a→next)
    if (a→tip→dp > d) d = a→tip→dp;
```

This code is used in section 13.

15. ⟨Set d to the maximum depth of an output of g 15⟩ ≡

```
  d = 0;
  for (a = g→outs; a; a = a→next)
    if (¬is_boolean(a→tip) ∧ a→tip→dp > d) d = a→tip→dp;
```

This code is used in section 13.

16. Index. Finally, here's a list that shows where the identifiers of this program are defined and used.

May 19, 2018 at 02:30

MULTIPLY

Preliminary work on the Stanford GraphBase project was supported in part by National Science Foundation grant CCR-86-10181.