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

1. Introduction. This GraphBase module provides six external subroutines:

risc, a routine that creates a directed acyclic graph based on the logic of a simple RISC computer;
prod, a routine that creates a directed acyclic graph based on the logic of parallel multiplication circuits;
print_gates, a routine that outputs a symbolic representation of such directed acyclic graphs;
gate_eval, a routine that evaluates such directed acyclic graphs by assigning boolean values to each gate;
partial_gates, a routine that extracts a subgraph by assigning random values to some of the input gates;
run_risc, a routine that can be used to play with the output of *risc*.

Examples of the use of these routines can be found in the demo programs TAKE_RISC and MULTIPLY.

```
⟨gb_gates.h 1⟩ ≡
#define print_gates p_gates   /* abbreviation for Procrustean linkers */
  extern Graph *risc();   /* make a network for a microprocessor */
  extern Graph *prod();   /* make a network for high-speed multiplication */
  extern void print_gates();   /* write a network to standard output file */
  extern long gate_eval();   /* evaluate a network */
  extern Graph *partial_gates();   /* reduce network size */
  extern long run_risc();   /* simulate the microprocessor */
  extern unsigned long risc_state[];   /* the output of run_risc */
```

See also sections 2 and 50.

2. The directed acyclic graphs produced by GB_GATES are GraphBase graphs with special conventions related to logical networks. Each vertex represents a gate of a network, and utility field *val* is a boolean value associated with that gate. Utility field *typ* is an ASCII code that tells what kind of gate is present:

- `'I'` denotes an input gate, whose value is specified externally.
- `'&'` denotes an `AND` gate, whose value is the logical AND of two or more previous gates (namely, 1 if all those gates are 1, otherwise 0).
- `'|'` denotes an `OR` gate, whose value is the logical OR of two or more previous gates (namely, 0 if all those gates are 0, otherwise 1).
- `'^'` denotes an `XOR` gate, whose value is the logical EXCLUSIVE-OR of two or more previous gates (namely, their sum modulo 2).
- `'~'` denotes an inverter, whose value is the logical complement of the value of a single previous gate.
- `'L'` denotes a latch, whose value depends on past history; it is the value that was assigned to a subsequent gate when the network was most recently evaluated. Utility field *alt* points to that subsequent gate.

Latches can be used to include "state" information in a circuit; for example, they correspond to registers of the RISC machine constructed by *risc*. The *prod* procedure does not use latches.

The vertices of the directed acyclic graph appear in a special "topological" order convenient for evaluation: All the input gates come first, followed by all the latches; then come the other types of gates, whose values are computed from their predecessors. The arcs of the graph run from each gate to its arguments, and all arguments to a gate precede that gate.

If g points to such a graph of gates, the utility field *g→outs* points to a list of **Arc** records, denoting "outputs" that might be used in certain applications. For example, the outputs of the graphs created by *prod* correspond to the bits of the product of the numbers represented in the input gates.

A special convention is used so that the routines will support partial evaluation: The *tip* fields in the output list either point to a vertex or hold one of the constant values 0 or 1 when regarded as an unsigned long integer.

```
#define val x.I     /* the field containing a boolean value */
#define typ y.I     /* the field containing the gate type */
#define alt z.V     /* the field pointing to another related gate */
#define outs zz.A    /* the field pointing to the list of output gates */
#define is_boolean(v) ((unsigned long) (v) ≤ 1)     /* is a tip field constant? */
#define the_boolean(v) ((long) (v))     /* if so, this is its value */
#define tip_value(v) (is_boolean(v) ? the_boolean(v) : (v)→val)
#define AND '&'
#define OR '|'
#define NOT '~'
#define XOR '^'
```

```
⟨gb_gates.h 1⟩ +≡
#define val x.I     /* the definitions are repeated in the header file */
#define typ y.I
#define alt z.V
#define outs zz.A
#define is_boolean(v)     ((unsigned long) (v) ≤ 1)
#define the_boolean(v)     ((long) (v))
#define tip_value(v)     (is_boolean(v) ? the_boolean(v) : (v)→val)
#define AND     '&'
#define OR      '|'
#define NOT     '~'
#define XOR     '^'
```

3. Let's begin with the *gate_eval* procedure, because it is quite simple and because it illustrates the conventions just explained. Given a gate graph g and optional pointers *in_vec* and *out_vec*, the procedure *gate_eval* will assign values to each gate of g. If *in_vec* is non-null, it should point to a string of characters, each `'0'` or `'1'`, that will be assigned to the first gates of the network, in order; otherwise *gate_eval* assumes that all input gates have already received appropriate values and it will not change them. New values are computed for each gate after the bits of *in_vec* have been consumed.

If *out_vec* is non-null, it should point to a memory area capable of receiving $m+1$ characters, where m is the number of outputs of g; a string containing the respective output values will be deposited there.

If *gate_eval* encounters an unknown gate type, it terminates execution prematurely and returns the value -1. Otherwise it returns 0.

```
⟨ The gate_eval routine 3 ⟩ ≡
  long gate_eval(g, in_vec, out_vec)
      Graph *g;      /* graph with gates as vertices */
      char *in_vec;      /* string for input values, or Λ */
      char *out_vec;      /* string for output values, or Λ */
  { register Vertex *v;      /* the current vertex of interest */
    register Arc *a;      /* the current arc of interest */
    register char t;      /* boolean value being computed */
    if (¬g) return −2;      /* no graph supplied! */
    v = g→vertices;
    if (in_vec) ⟨ Read a sequence of input values from in_vec 4 ⟩;
    for ( ; v < g→vertices + g→n; v++) {
      switch (v→typ) {      /* branch on type of gate */
      case 'I': continue;      /* this input gate's value should be externally set */
      case 'L': t = v→alt→val; break;
    ⟨ Compute the value t of a classical logic gate 6 ⟩;
      default: return −1;      /* unknown gate type! */
      }
      v→val = t;      /* assign the computed value */
    }
    if (out_vec) ⟨ Store the sequence of output values in out_vec 5 ⟩;
    return 0;
  }
```

This code is used in section 7.

4. ⟨ Read a sequence of input values from *in_vec* 4 ⟩ ≡

```
  while (*in_vec ∧ v < g→vertices + g→n) (v++)→val = *in_vec++ − '0';
```

This code is used in section 3.

5. ⟨ Store the sequence of output values in *out_vec* 5 ⟩ ≡

```
  {
    for (a = g→outs; a; a = a→next) *out_vec++ = '0' + tip_value(a→tip);
    *out_vec = 0;      /* terminate the string */
  }
```

This code is used in section 3.

6. ⟨ Compute the value t of a classical logic gate 6 ⟩ ≡

```
case AND: t = 1;
  for (a = v→arcs; a; a = a→next) t &= a→tip→val;
  break;
case OR: t = 0;
  for (a = v→arcs; a; a = a→next) t |= a→tip→val;
  break;
case XOR: t = 0;
  for (a = v→arcs; a; a = a→next) t ⊕= a→tip→val;
  break;
case NOT: t = 1 − v→arcs→tip→val;
  break;
```

This code is used in section 3.

7. Here now is an outline of the entire GB_GATES module, as seen by the C compiler:

```
#include "gb_flip.h"    /* we will use the GB_FLIP routines for random numbers */
#include "gb_graph.h"   /* and we will use the GB_GRAPH data structures */
  ⟨ Preprocessor definitions ⟩
  ⟨ Private variables 12 ⟩
  ⟨ Global variables 48 ⟩
  ⟨ Internal subroutines 11 ⟩
  ⟨ The gate_eval routine 3 ⟩
  ⟨ The print_gates routine 49 ⟩
  ⟨ The risc routine 8 ⟩
  ⟨ The run_risc routine 43 ⟩
  ⟨ The prod routine 66 ⟩
  ⟨ The partial_gates routine 84 ⟩
```

8. The RISC netlist. The subroutine call *risc*(*regs*) creates a gate graph having *regs* registers; the value of *regs* must be between 2 and 16, inclusive, otherwise *regs* is set to 16. This gate graph describes the circuitry for a small RISC computer, defined below. The total number of gates turns out to be $1400 + 115 * regs$; thus it lies between 1630 (when $regs = 2$) and 3240 (when $regs = 16$). EXCLUSIVE-OR gates are not used; the effect of xoring is obtained where needed by means of ANDs, ORs, and inverters.

If *risc* cannot do its thing, it returns Λ (NULL) and sets *panic_code* to indicate the problem. Otherwise *risc* returns a pointer to the graph.

```
#define panic(c) { panic_code = c; gb_trouble_code = 0; return Λ; }
```

```
⟨ The risc routine 8 ⟩ ≡
  Graph *risc(regs)
      unsigned long regs;    /* number of registers supported */
  { ⟨ Local variables for risc 9 ⟩
    ⟨ Initialize new_graph to an empty graph of the appropriate size 16 ⟩;
    ⟨ Add the RISC data to new_graph 17 ⟩;
    if (gb_trouble_code) {
      gb_recycle(new_graph);
      panic(alloc_fault);    /* oops, we ran out of memory somewhere back there */
    }
    return new_graph;
  }
```

This code is used in section 7.

9.

```
⟨ Local variables for risc 9 ⟩ ≡
  Graph *new_graph;    /* the graph constructed by risc */
  register long k, r;    /* all-purpose indices */
```

See also sections 18, 20, 25, 28, 33, 37, and 40.

This code is used in section 8.

10. This RISC machine works with 16-bit registers and 16-bit data words. It cannot write into memory, but it assumes the existence of an external read-only memory. The circuit has 16 outputs, representing the 16 bits of a memory address register. It also has 17 inputs, the last 16 of which are supposed to be set to the contents of the memory address computed on the previous cycle. Thus we can run the machine by accessing memory between calls of *gate_eval*. The first input bit, called `RUN`, is normally set to 1; if it is 0, the other inputs are effectively ignored and all registers and outputs will be cleared to 0. Input bits for the memory appear in "little-endian order," that is, least significant bit first; but the output bits for the memory address register appear in "big-endian order," most significant bit first.

Words read from memory are interpreted as instructions having the following format:

`DST`	`MOD`	`OP`	`A`	`SRC`
15 14 13 12	11 10 9 8	7 6	5 4	3 2 1 0

The `SRC` and `A` fields specify a "source" value. If `A` $= 0$, the source is `SRC`, treated as a 16-bit signed number between -8 and $+7$ inclusive. If `A` $= 1$, the source is the contents of register `DST` plus the (signed) value of `SRC`. If `A` $= 2$, the source is the contents of register `SRC`. And if `A` $= 3$, the source is the contents of the memory location whose address is the contents of register `SRC`. Thus, for example, if `DST` $= 3$ and `SRC` $= 10$, and if `r3` contains 17 while `r10` contains 1009, the source value will be -6 if `A` $= 0$, or $17 - 6 = 11$ if `A` $= 1$, or 1009 if `A` $= 2$, or the contents of memory location 1009 if `A` $= 3$.

The `DST` field specifies the number of the destination register. This register receives a new value based on its previous value and the source value, as prescribed by the operation defined in the `OP` and `MOD` fields. For example, when `OP` $= 0$, a general logical operation is performed, as follows: Suppose the bits of `MOD` are called $\mu_{11}\mu_{10}\mu_{01}\mu_{00}$ from left to right. Then if the kth bit of the destination register currently is equal to i and the kth bit of the source value is equal to j, the general logical operator changes the kth bit of the destination register to μ_{ij}. If the `MOD` bits are, for example, 1010, the source value is simply copied to the destination register; if `MOD` $= 0110$, an exclusive-or is done; if `MOD` $= 0011$, the destination register is complemented and the source value is effectively ignored.

The machine contains four status bits called `S` (sign), `N` (nonzero), `K` (carry), and `V` (overflow). Every general logical operation sets `S` equal to the sign of the new result transferred to the destination register; this is bit 15, the most significant bit. A general logical operation also sets `N` to 1 if any of the other 15 bits are 1, to 0 if all of the other bits are 0. Thus `S` and `N` both become zero if and only if the new result is entirely zero. Logical operations do not change the values of `K` and `V`; the latter are affected only by the arithmetic operations described below.

The status of the `S` and `N` bits can be tested by using the conditional load operator, `OP` $= 2$: This operation loads the source value into the destination register if and only if `MOD` bit $\mu_{ij} = 1$, where i and j are the current values of `S` and `N`, respectively. For example, if `MOD` $= 0011$, the source value is loaded if and only if `S` $= 0$, which means that the last value affecting `S` and `N` was greater than or equal to zero. If `MOD` $= 1111$, loading is always done; this option provides a way to move source to destination without affecting `S` or `N`.

A second conditional load operator, `OP` $= 3$, is similar, but it is used for testing the status of `K` and `V` instead of `S` and `N`. For example, a command having `MOD` $= 1010$, `OP` $= 3$, `A` $= 1$, and `SRC` $= 1$ adds the current overflow bit to the destination register. (Please take a moment to understand why this is true.)

We have now described all the operations except those that are performed when `OP` $= 1$. As you might expect, our machine is able to do rudimentary arithmetic. The general addition and subtraction operators belong to this final case, together with various shift operators, depending on the value of `MOD`.

Eight of the `OP` $= 1$ operations set the destination register to a shifted version of the source value: `MOD` $= 0$ means "shift left 1," which is equivalent to multiplying the source by 2; `MOD` $= 1$ means "cyclic shift left 1," which is the same except that it also adds the previous sign bit to the result; `MOD` $= 2$ means "shift left 4," which is equivalent to multiplying by 16; `MOD` $= 3$ means "cyclic shift left 4"; `MOD` $= 4$ means "shift right 1," which is equivalent to dividing the source by 2 and rounding down to the next lower integer if there was a remainder; `MOD` $= 5$ means "unsigned shift right 1," which is the same except that the most significant bit is always set to zero instead of retaining the previous sign; `MOD` $= 6$ means "shift right 4," which is equivalent to dividing the source by 16 and rounding down; `MOD` $= 7$ means "unsigned shift right 4." Each of these shift operations affects S and N, as in the case of logical operations. They also affect K and V, as follows: Shifting left sets K to 1 if and only if at least one of the bits shifted off the left was nonzero, and sets V to 1 if and only if the corresponding multiplication would cause overflow. Shifting right 1 sets K to the value of the bit shifted out, and sets V to 0; shifting right 4 sets K to the value of the last bit shifted out, and sets V to the logical OR of the other three lost bits. The same values of K and V arise from cyclic or unsigned shifts as from ordinary shifts.

When OP $= 1$ and MOD $= 8$, the source value is added to the destination register. This sets S, N, and V as you would expect; and it sets K to the carry you would get if you were treating the operands as 16-bit unsigned integers. Another addition operation, having MOD $= 9$, is similar, but the current value of K is also added to the result; in this case, the new value of N will be zero if and only if the 15 non-sign bits of the result are zero and the previous values of S and N were also zero. This means that you can use the first addition operation on the lower halves of a 32-bit number and the second operation on the upper halves, thereby obtaining a correct 32-bit result, with appropriate sign, nonzero, carry, and overflow bits set. Higher precision (48 bits, 64 bits, etc.) can be obtained in a similar way.

When OP $= 1$ and MOD $= 10$, the source value is subtracted from the destination register. Again, S, N, K, and V are set; the K value in this case represents the "borrow" bit. An auxiliary subtraction operation, having MOD $= 11$, subtracts also the current value of K, thereby allowing for correct 32-bit subtraction.

The operations for OP $= 1$ and MOD $= 12$, 13, and 14 are "reserved for future expansion." Actually they will never change, however, since this RISC chip is purely academic. If you check out the logic below, you will find that they simply set the destination register and the four status bits all to zero.

A final operation, called JUMP, will be explained momentarily. It has OP $= 1$ and MOD $= 15$. It does not affect S, N, K, or V.

If the RISC is made with fewer than 16 registers, the higher-numbered ones will effectively contain zero whenever their values are fetched. But if you use them as destination registers, you will set S, N, K, and V as if actual numbers were being stored.

Register 0 is different from the other 15 registers: It is the location of the current instruction. Therefore if you change the contents of register 0, you are changing the control flow of the program. If you do not change register 0, it automatically increases by 1.

Special treatment occurs when A $= 3$ and SRC $= 0$. In such a case, the normal rules given above say that the source value should be the contents of the memory location specified by register 0. But that memory location holds the current instruction; so the machine uses the *following* location instead, as a 16-bit source operand. If the contents of register 0 are not changed by such a two-word instruction, register 0 will increase by 2 instead of 1.

We have now discussed everything about the machine except the operation of the JUMP command. This command moves the source value to register 0, thereby changing the flow of control. Furthermore, if DST $\neq 0$, it also sets register DST to the location of the instruction following the JUMP. Assembly language programmers will recognize this as a convenient way to jump to a subroutine.

Example programs can be found in the TAKE_RISC module, which includes a simple subroutine for multiplication and division.

11. A few auxiliary functions will ameliorate the task of constructing the RISC logic. First comes a routine that "christens" a new gate, assigning it a name and a type. The name is constructed from a prefix and a serial number, where the prefix indicates the current portion of logic being created.

```
⟨ Internal subroutines 11 ⟩ ≡
  static Vertex *new_vert(t)
       char t;     /* the type of the new gate */
  { register Vertex *v;
    v = next_vert++;
    if (count < 0) v→name = gb_save_string(prefix);
    else {
      sprintf(name_buf, "%s%ld", prefix, count);
      v→name = gb_save_string(name_buf);
      count++;
    }
    v→typ = t;
    return v;
  }
```

See also sections 13, 14, 15, 38, and 51.

This code is used in section 7.

12. **#define** *start_prefix*(*s*) *strcpy*(*prefix*, *s*); *count* = 0
#define *numeric_prefix*(*a*, *b*) *sprintf*(*prefix*, "%c%ld:", *a*, *b*); *count* = 0;

```
⟨ Private variables 12 ⟩ ≡
  static Vertex *next_vert;     /* the first vertex not yet assigned a name */
  static char prefix[5];     /* prefix string for vertex names */
  static long count;     /* serial number for vertex names */
  static char name_buf[100];     /* place to form vertex names */
```

This code is used in section 7.

13. Here are some trivial routines to create gates with 2, 3, or more arguments. The arcs from such a gate to its inputs are assigned length 100. Other routines, defined below, assign length 1 to the arc between an inverter and its unique input. This convention makes the lengths of shortest paths in the resulting network a bit more interesting than they would otherwise be.

```
#define DELAY 100L
⟨Internal subroutines 11⟩ +≡
  static Vertex *make2(t, v1, v2)
      char t;   /* the type of the new gate */
      Vertex *v1, *v2;
  { register Vertex *v = new_vert(t);
    gb_new_arc(v, v1, DELAY);
    gb_new_arc(v, v2, DELAY);
    return v;
  }
  static Vertex *make3(t, v1, v2, v3)
      char t;   /* the type of the new gate */
      Vertex *v1, *v2, *v3;
  { register Vertex *v = new_vert(t);
    gb_new_arc(v, v1, DELAY);
    gb_new_arc(v, v2, DELAY);
    gb_new_arc(v, v3, DELAY);
    return v;
  }
  static Vertex *make4(t, v1, v2, v3, v4)
      char t;   /* the type of the new gate */
      Vertex *v1, *v2, *v3, *v4;
  { register Vertex *v = new_vert(t);
    gb_new_arc(v, v1, DELAY);
    gb_new_arc(v, v2, DELAY);
    gb_new_arc(v, v3, DELAY);
    gb_new_arc(v, v4, DELAY);
    return v;
  }
  static Vertex *make5(t, v1, v2, v3, v4, v5)
      char t;   /* the type of the new gate */
      Vertex *v1, *v2, *v3, *v4, *v5;
  { register Vertex *v = new_vert(t);
    gb_new_arc(v, v1, DELAY);
    gb_new_arc(v, v2, DELAY);
    gb_new_arc(v, v3, DELAY);
    gb_new_arc(v, v4, DELAY);
    gb_new_arc(v, v5, DELAY);
    return v;
  }
```

14. We use utility field $w.V$ to store a pointer to the complement of a gate, if that complement has been formed. This trick prevents the creation of excessive gates that are equivalent to each other. The following subroutine returns a pointer to the complement of a given gate.

```
#define bar w.V    /* field pointing to complement, if known to exist */
#define even_comp(s, v) ((s) & 1 ? v : comp(v))
⟨ Internal subroutines 11 ⟩ +≡
  static Vertex *comp(v)
      Vertex *v;
  { register Vertex *u;
    if (v→bar) return v→bar;
    u = next_vert++;
    u→bar = v; v→bar = u;
    sprintf(name_buf, "%s~", v→name);
    u→name = gb_save_string(name_buf);
    u→typ = NOT;
    gb_new_arc(u, v, 1L);
    return u;
  }
```

15. To create a gate for the EXCLUSIVE-OR of two arguments, we can either construct the OR of two ANDs or the AND of two ORs. We choose the former alternative:

```
⟨ Internal subroutines 11 ⟩ +≡
  static Vertex *make_xor(u, v)
      Vertex *u, *v;
  { register Vertex *t1, *t2;
    t1 = make2(AND, u, comp(v));
    t2 = make2(AND, comp(u), v);
    return make2(OR, t1, t2);
  }
```

16. OK, let's get going.

```
⟨ Initialize new_graph to an empty graph of the appropriate size 16 ⟩ ≡
  if (regs < 2 ∨ regs > 16) regs = 16;
  new_graph = gb_new_graph(1400 + 115 * regs);
  if (new_graph ≡ Λ) panic(no_room);    /* out of memory before we're even started */
  sprintf(new_graph→id, "risc(%lu)", regs);
  strcpy(new_graph→util_types, "ZZZIIVZZZZZZZA");
  next_vert = new_graph→vertices;
```

This code is used in section 8.

17.
```
⟨ Add the RISC data to new_graph 17 ⟩ ≡
  ⟨ Create the inputs and latches 19 ⟩;
  ⟨ Create gates for instruction decoding 21 ⟩;
  ⟨ Create gates for fetching the source value 22 ⟩;
  ⟨ Create gates for the general logic operation 26 ⟩;
  ⟨ Create gates for the conditional load operations 27 ⟩;
  ⟨ Create gates for the arithmetic operations 41 ⟩;
  ⟨ Create gates that bring everything together properly 29 ⟩;
  if (next_vert ≠ new_graph→vertices + new_graph→n) panic(impossible);
      /* oops, we miscounted; this should be impossible */
```

This code is used in section 8.

18. Internal names will make it convenient to refer to the most important gates. Here are the names of inputs and latches.

```
⟨ Local variables for risc 9 ⟩ +≡
  Vertex *run_bit;    /* the RUN input */
  Vertex *mem[16];    /* 16 bits of input from read-only memory */
  Vertex *prog;    /* first of 10 bits in the program register */
  Vertex *sign;    /* the latched value of S */
  Vertex *nonzero;    /* the latched value of N */
  Vertex *carry;    /* the latched value of K */
  Vertex *overflow;    /* the latched value of V */
  Vertex *extra;    /* latched status bit: are we doing an extra memory cycle? */
  Vertex *reg[16];    /* the least-significant bit of a given register */
```

19. **#define** *first_of*(*n*, *t*) *new_vert*(*t*); **for** ($k = 1$; $k < n$; k++) *new_vert*(*t*);

```
⟨ Create the inputs and latches 19 ⟩ ≡
  strcpy(prefix, "RUN"); count = −1; run_bit = new_vert('I');
  start_prefix("M"); for (k = 0; k < 16; k++) mem[k] = new_vert('I');
  start_prefix("P"); prog = first_of(10, 'L');
  strcpy(prefix, "S"); count = −1; sign = new_vert('L');
  strcpy(prefix, "N"); nonzero = new_vert('L');
  strcpy(prefix, "K"); carry = new_vert('L');
  strcpy(prefix, "V"); overflow = new_vert('L');
  strcpy(prefix, "X"); extra = new_vert('L');
  for (r = 0; r < regs; r++) {
    numeric_prefix('R', r);
    reg[r] = first_of(16, 'L');
  }
```

This code is used in section 17.

20. The order of evaluation of function arguments is not defined in C, so we introduce a few macros that force left-to-right order.

```
#define do2(result, t, v1, v2)
        { t1 = v1; t2 = v2;
          result = make2(t, t1, t2); }
#define do3(result, t, v1, v2, v3)
        { t1 = v1; t2 = v2; t3 = v3;
          result = make3(t, t1, t2, t3); }
#define do4(result, t, v1, v2, v3, v4)
        { t1 = v1; t2 = v2; t3 = v3; t4 = v4;
          result = make4(t, t1, t2, t3, t4); }
#define do5(result, t, v1, v2, v3, v4, v5)
        { t1 = v1; t2 = v2; t3 = v3; t4 = v4; t5 = v5;
          result = make5(t, t1, t2, t3, t4, t5); }
⟨Local variables for risc 9⟩ +≡
  Vertex *t1, *t2, *t3, *t4, *t5;    /* temporary holds to force evaluation order */
  Vertex *tmp[16];    /* additional holding places for partial results */
  Vertex *imm;    /* is the source value immediate (a given constant)? */
  Vertex *rel;    /* is the source value relative to the current destination register? */
  Vertex *dir;    /* should the source value be fetched directly from a source register? */
  Vertex *ind;    /* should the source value be fetched indirectly from memory? */
  Vertex *op;    /* least significant bit of OP */
  Vertex *cond;    /* most significant bit of OP */
  Vertex *mod[4];    /* the MOD bits */
  Vertex *dest[4];    /* the DEST bits */
```

21. The sixth line of the program here can be translated into the logic equation

$$op = (extra \land prog) \lor (\overline{extra} \land mem[6])\,.$$

Once you see why, you'll be able to read the rest of this curious code.

```
⟨Create gates for instruction decoding 21⟩ ≡
  start_prefix("D");
  do3(imm, AND, comp(extra), comp(mem[4]), comp(mem[5]));    /* A = 0 */
  do3(rel, AND, comp(extra), mem[4], comp(mem[5]));    /* A = 1 */
  do3(dir, AND, comp(extra), comp(mem[4]), mem[5]);    /* A = 2 */
  do3(ind, AND, comp(extra), mem[4], mem[5]);    /* A = 3 */
  do2(op, OR, make2(AND, extra, prog), make2(AND, comp(extra), mem[6]));
  do2(cond, OR, make2(AND, extra, prog + 1), make2(AND, comp(extra), mem[7]));
  for (k = 0; k < 4; k++) {
    do2(mod[k], OR, make2(AND, extra, prog + 2 + k), make2(AND, comp(extra), mem[8 + k]));
    do2(dest[k], OR, make2(AND, extra, prog + 6 + k), make2(AND, comp(extra), mem[12 + k]));
  }
```

This code is used in section 17.

22. ⟨ Create gates for fetching the source value 22 ⟩ ≡

```
start_prefix("F");
⟨ Set old_dest to the present value of the destination register 23 ⟩;
⟨ Set old_src to the present value of the source register 24 ⟩;
⟨ Set inc_dest to old_dest plus SRC 39 ⟩;
for (k = 0; k < 16; k++)
  do4(source[k], OR, make2(AND, imm, mem[k < 4 ? k : 3]), make2(AND, rel, inc_dest[k]),
      make2(AND, dir, old_src[k]), make2(AND, extra, mem[k]));
```

This code is used in section 17.

23. Here and in the immediately following section we create OR gates *old_dest*[*k*] and *old_src*[*k*] that might have as many as 16 inputs. (The actual number of inputs is *regs*.) All other gates in the network will have at most five inputs.

⟨ Set *old_dest* to the present value of the destination register 23 ⟩ ≡

```
for (r = 0; r < regs; r++)
  do4(dest_match[r], AND, even_comp(r, dest[0]), even_comp(r ≫ 1, dest[1]),
      even_comp(r ≫ 2, dest[2]), even_comp(r ≫ 3, dest[3]));
for (k = 0; k < 16; k++) {
  for (r = 0; r < regs; r++)
    tmp[r] = make2(AND, dest_match[r], reg[r] + k);
  old_dest[k] = new_vert(OR);
  for (r = 0; r < regs; r++) gb_new_arc(old_dest[k], tmp[r], DELAY);
}
```

This code is used in section 22.

24. ⟨ Set *old_src* to the present value of the source register 24 ⟩ ≡

```
for (k = 0; k < 16; k++) {
  for (r = 0; r < regs; r++)
    do5(tmp[r], AND, reg[r] + k, even_comp(r, mem[0]), even_comp(r ≫ 1, mem[1]), even_comp(r ≫ 2,
        mem[2]), even_comp(r ≫ 3, mem[3]));
  old_src[k] = new_vert(OR);
  for (r = 0; r < regs; r++) gb_new_arc(old_src[k], tmp[r], DELAY);
}
```

This code is used in section 22.

25. ⟨ Local variables for *risc* 9 ⟩ +≡

```
Vertex *dest_match[16];  /* dest_match[r] ≡ 1 iff DST = r */
Vertex *old_dest[16];  /* contents of destination register before operation */
Vertex *old_src[16];  /* contents of source register before operation */
Vertex *inc_dest[16];  /* old_dest plus the SRC field */
Vertex *source[16];  /* source value for the operation */
Vertex *log[16];  /* result of general logic operation */
Vertex *shift[18];  /* result of shift operation, with carry and overflow */
Vertex *sum[18];  /* old_dest plus source plus optional carry */
Vertex *diff[18];  /* old_dest minus source minus optional borrow */
Vertex *next_loc[16];  /* contents of register 0, plus 1 */
Vertex *next_next_loc[16];  /* contents of register 0, plus 2 */
Vertex *result[18];  /* result of operating on old_dest and source */
```

26. ⟨ Create gates for the general logic operation 26 ⟩ ≡

```
start_prefix("L");
for (k = 0; k < 16; k++)
  do4(log[k], OR,
      make3(AND, mod[0], comp(old_dest[k]), comp(source[k])),
      make3(AND, mod[1], comp(old_dest[k]), source[k]),
      make3(AND, mod[2], old_dest[k], comp(source[k])),
      make3(AND, mod[3], old_dest[k], source[k]));
```

This code is used in section 17.

27. ⟨ Create gates for the conditional load operations 27 ⟩ ≡

```
start_prefix("C");
do4(tmp[0], OR,
    make3(AND, mod[0], comp(sign), comp(nonzero)),
    make3(AND, mod[1], comp(sign), nonzero),
    make3(AND, mod[2], sign, comp(nonzero)),
    make3(AND, mod[3], sign, nonzero));
do4(tmp[1], OR,
    make3(AND, mod[0], comp(carry), comp(overflow)),
    make3(AND, mod[1], comp(carry), overflow),
    make3(AND, mod[2], carry, comp(overflow)),
    make3(AND, mod[3], carry, overflow));
do3(change, OR, comp(cond), make2(AND, tmp[0], comp(op)), make2(AND, tmp[1], op));
```

This code is used in section 17.

28. ⟨ Local variables for *risc* 9 ⟩ +≡

```
Vertex *change;    /* is the destination register supposed to change? */
```

29. Hardware is like software except that it performs all the operations all the time and then selects only the results it needs. (If you think about it, this is a profound observation about economics, society, and nature. Gosh.)

⟨ Create gates that bring everything together properly 29 ⟩ ≡

```
start_prefix("Z");
⟨ Create gates for the next_loc and next_next_loc bits 30 ⟩;
⟨ Create gates for the result bits 31 ⟩;
⟨ Create gates for the new values of registers 1 to regs 34 ⟩;
⟨ Create gates for the new values of S, N, K, and V 35 ⟩;
⟨ Create gates for the new values of the program register and extra 32 ⟩;
⟨ Create gates for the new values of register 0 and the memory address register 36 ⟩;
```

This code is used in section 17.

30. ⟨ Create gates for the *next_loc* and *next_next_loc* bits 30 ⟩ ≡

```
next_loc[0] = comp(reg[0]); next_next_loc[0] = reg[0];
next_loc[1] = make_xor(reg[0] + 1, reg[0]); next_next_loc[1] = comp(reg[0] + 1);
for (t5 = reg[0] + 1, k = 2; k < 16; t5 = make2(AND, t5, reg[0] + k++)) {
  next_loc[k] = make_xor(reg[0] + k, make2(AND, reg[0], t5));
  next_next_loc[k] = make_xor(reg[0] + k, t5);
}
```

This code is used in section 29.

31. ⟨ Create gates for the *result* bits 31 ⟩ ≡

```
jump = make5(AND, op, mod[0], mod[1], mod[2], mod[3]);    /* assume cond = 0 */
for (k = 0; k < 16; k++) {
  do5(result[k], OR,
      make2(AND, comp(op), log[k]),
      make2(AND, jump, next_loc[k]),
      make3(AND, op, comp(mod[3]), shift[k]),
      make5(AND, op, mod[3], comp(mod[2]), comp(mod[1]), sum[k]),
      make5(AND, op, mod[3], comp(mod[2]), mod[1], diff[k]));
  do2(result[k], OR,
      make3(AND, cond, change, source[k]),
      make2(AND, comp(cond), result[k]));
}
for (k = 16; k < 18; k++)    /* carry and overflow bits of the result */
  do3(result[k], OR,
      make3(AND, op, comp(mod[3]), shift[k]),
      make5(AND, op, mod[3], comp(mod[2]), comp(mod[1]), sum[k]),
      make5(AND, op, mod[3], comp(mod[2]), mod[1], diff[k]));
```

This code is used in section 29.

32. The program register *prog* and the *extra* bit are needed for the case when we must spend an extra cycle to fetch a word from memory. On the first cycle, *ind* is true, so a "result" is calculated but not actually used. On the second cycle, *extra* is true.

A slight optimization has been introduced in order to make the circuit a bit more interesting: If a conditional load instruction occurs with indirect addressing and a false condition, the extra cycle is not taken. (The *next_next_loc* values were computed for this reason.)

```
#define latchit(u, latch) (latch)→alt = make2(AND, u, run_bit)
          /* u & run_bit is new value for latch */
```

⟨ Create gates for the new values of the program register and *extra* 32 ⟩ ≡

```
for (k = 0; k < 10; k++) latchit(mem[k + 6], prog + k);
do2(nextra, OR, make2(AND, ind, comp(cond)), make2(AND, ind, change));
latchit(nextra, extra);
nzs = make4(OR, mem[0], mem[1], mem[2], mem[3]);
nzd = make4(OR, dest[0], dest[1], dest[2], dest[3]);
```

This code is used in section 29.

33. ⟨ Local variables for *risc* 9 ⟩ +≡

```
Vertex *jump;    /* is this command a JUMP, assuming cond is false? */
Vertex *nextra;    /* must we take an extra cycle? */
Vertex *nzs;    /* is the SRC field nonzero? */
Vertex *nzd;    /* is the DST field nonzero? */
```

34. ⟨ Create gates for the new values of registers 1 to *regs* 34 ⟩ ≡

```
t5 = make2(AND, change, comp(ind));    /* should destination register change? */
for (r = 1; r < regs; r++) {
  t4 = make2(AND, t5, dest_match[r]);    /* should register r change? */
  for (k = 0; k < 16; k++) {
    do2(t3, OR, make2(AND, t4, result[k]), make2(AND, comp(t4), reg[r] + k));
    latchit(t3, reg[r] + k);
  }
}
```

This code is used in section 29.

35. ⟨ Create gates for the new values of S, N, K, and V 35 ⟩ ≡

```
do4 (t5, OR,
      make2 (AND, sign, cond),
      make2 (AND, sign, jump),
      make2 (AND, sign, ind),
      make4 (AND, result[15], comp (cond), comp (jump), comp (ind)));
latchit (t5, sign);
do4 (t5, OR,
      make4 (OR, result[0], result[1], result[2], result[3]),
      make4 (OR, result[4], result[5], result[6], result[7]),
      make4 (OR, result[8], result[9], result[10], result[11]),
      make4 (OR, result[12], result[13], result[14],
               make5 (AND, make2 (OR, nonzero, sign), op, mod[0], comp (mod[2]), mod[3])));
do4 (t5, OR,
      make2 (AND, nonzero, cond),
      make2 (AND, nonzero, jump),
      make2 (AND, nonzero, ind),
      make4 (AND, t5, comp (cond), comp (jump), comp (ind)));
latchit (t5, nonzero);
do5 (t5, OR,
      make2 (AND, overflow, cond),
      make2 (AND, overflow, jump),
      make2 (AND, overflow, comp (op)),
      make2 (AND, overflow, ind),
      make5 (AND, result[17], comp (cond), comp (jump), comp (ind), op));
latchit (t5, overflow);
do5 (t5, OR,
      make2 (AND, carry, cond),
      make2 (AND, carry, jump),
      make2 (AND, carry, comp (op)),
      make2 (AND, carry, ind),
      make5 (AND, result[16], comp (cond), comp (jump), comp (ind), op));
latchit (t5, carry);
```

This code is used in section 29.

36. As usual, we have left the hardest case for last, hoping that we will have learned enough tricks to handle it when the time of reckoning finally arrives.

The most subtle part of the logic here is perhaps the case of a JUMP command with A = 3. We want to increase register 0 by 1 during the first cycle of such a command, if SRC = 0, so that the *result* will be correct on the next cycle.

```
⟨ Create gates for the new values of register 0 and the memory address register 36 ⟩ ≡
  skip = make2(AND, cond, comp(change));   /* false conditional? */
  hop = make2(AND, comp(cond), jump);   /* JUMP command? */
  do4(normal, OR,
        make2(AND, skip, comp(ind)),
        make2(AND, skip, nzs),
        make3(AND, comp(skip), ind, comp(nzs)),
        make3(AND, comp(skip), comp(hop), nzd));
  special = make3(AND, comp(skip), ind, nzs);
  for (k = 0; k < 16; k++) {
    do4(t5, OR,
          make2(AND, normal, next_loc[k]),
          make4(AND, skip, ind, comp(nzs), next_next_loc[k]),
          make3(AND, hop, comp(ind), source[k]),
          make5(AND, comp(skip), comp(hop), comp(ind), comp(nzd), result[k]));
    do2(t4, OR,
          make2(AND, special, reg[0] + k),
          make2(AND, comp(special), t5));
    latchit(t4, reg[0] + k);
    do2(t4, OR,
          make2(AND, special, old_src[k]),
          make2(AND, comp(special), t5));
    { register Arc *a = gb_virgin_arc();
      a→tip = make2(AND, t4, run_bit);
      a→next = new_graph→outs;
      new_graph→outs = a;   /* pointer to memory address bit */
    }
  }   /* arcs for output bits will appear in big-endian order */
```

This code is used in section 29.

37. ⟨ Local variables for *risc* 9 ⟩ +≡

```
  Vertex *skip;   /* are we skipping a conditional load operation? */
  Vertex *hop;   /* are we doing a JUMP? */
  Vertex *normal;   /* is this a case where register 0 is simply incremented? */
  Vertex *special;
    /* is this a case where register 0 and the memory address register will not coincide? */
```

38. Serial addition. We haven't yet specified the parts of *risc* that deal with addition and subtraction; somehow, those parts wanted to be separate from the rest. To complete our mission, we will use subroutine calls of the form '*make_adder*$(n, x, y, z,$ *carry*, *add*$)$', where x and y are n-bit arrays of input gates and z is an $(n+1)$-bit array of output gates. If *add* $\neq 0$, the subroutine computes $x+y$, otherwise it computes $x-y$. If *carry* $\neq 0$, the *carry* gate is effectively added to y before the operation.

A simple n-stage serial scheme, which reduces the problem of n-bit addition to $(n-1)$-bit addition, is adequate for our purposes here. (A parallel adder, which gains efficiency by reducing the problem size from n to n/ϕ, can be found in the *prod* routine below.)

The handy identity $x - y = \overline{\overline{x} + y}$ is used to reduce subtraction to addition.

```
⟨Internal subroutines 11⟩ +≡
  static void make_adder(n, x, y, z, carry, add)
        unsigned long n;      /* number of bits */
        Vertex *x[], *y[];      /* input gates */
        Vertex *z[];      /* output gates */
        Vertex *carry;      /* add this to y, unless it's null */
        char add;      /* should we add or subtract? */
  { register long k;
    Vertex *t1, *t2, *t3, *t4;      /* temporary storage used by do4 */
    if (¬carry) {
      z[0] = make_xor(x[0], y[0]);
      carry = make2(AND, even_comp(add, x[0]), y[0]);
      k = 1;
    } else k = 0;
    for ( ; k < n; k++) {
      comp(x[k]); comp(y[k]); comp(carry);      /* generate inverse gates */
      do4(z[k], OR,
            make3(AND, x[k], comp(y[k]), comp(carry)),
            make3(AND, comp(x[k]), y[k], comp(carry)),
            make3(AND, comp(x[k]), comp(y[k]), carry),
            make3(AND, x[k], y[k], carry));
      do3(carry, OR,
            make2(AND, even_comp(add, x[k]), y[k]),
            make2(AND, even_comp(add, x[k]), carry),
            make2(AND, y[k], carry));
    }
    z[n] = carry;
  }
```

39. OK, now we can add. What good does that do us? In the first place, we need a 4-bit adder to compute the least significant bits of *old_dest* + SRC. The other 12 bits of that sum are simpler.

```
⟨ Set inc_dest to old_dest plus SRC 39 ⟩ ≡
  make_adder(4_L, old_dest, mem, inc_dest, Λ, 1);
  up = make2(AND, inc_dest[4], comp(mem[3]));      /* remaining bits must increase */
  down = make2(AND, comp(inc_dest[4]), mem[3]);     /* remaining bits must decrease */
  for (k = 4; ; k++) {
    comp(up); comp(down);
    do3(inc_dest[k], OR,
        make2(AND, comp(old_dest[k]), up),
        make2(AND, comp(old_dest[k]), down),
        make3(AND, old_dest[k], comp(up), comp(down)));
    if (k < 15) {
      up = make2(AND, up, old_dest[k]);
      down = make2(AND, down, comp(old_dest[k]));
    } else break;
  }
```

This code is used in section 22.

40. ⟨ Local variables for *risc* 9 ⟩ +≡

```
  Vertex *up, *down;      /* gates used when computing inc_dest */
```

41. In the second place, we need a 16-bit adder and a 16-bit subtracter for the four addition/subtraction commands.

```
⟨ Create gates for the arithmetic operations 41 ⟩ ≡
  start_prefix("A");
  ⟨ Create gates for the shift operations 42 ⟩;
  make_adder(16_L, old_dest, source, sum, make2(AND, carry, mod[0]), 1);      /* adder */
  make_adder(16_L, old_dest, source, diff, make2(AND, carry, mod[0]), 0);     /* subtracter */
  do2(sum[17], OR,
      make3(AND, old_dest[15], source[15], comp(sum[15])),
      make3(AND, comp(old_dest[15]), comp(source[15]), sum[15]));      /* overflow */
  do2(diff[17], OR,
      make3(AND, old_dest[15], comp(source[15]), comp(diff[15])),
      make3(AND, comp(old_dest[15]), source[15], diff[15]));      /* overflow */
```

This code is used in section 17.

42. ⟨ Create gates for the shift operations 42 ⟩ ≡

```
for (k = 0; k < 16; k++)
  do4 (shift[k], OR,
      (k ≡ 0 ? make4 (AND, source[15], mod[0], comp (mod[1]), comp (mod[2])) :
                make3 (AND, source[k − 1], comp (mod[1]), comp (mod[2]))),
      (k < 4 ? make4 (AND, source[k + 12], mod[0], mod[1], comp (mod[2])) :
                make3 (AND, source[k − 4], mod[1], comp (mod[2]))),
      (k ≡ 15 ? make4 (AND, source[15], comp (mod[0]), comp (mod[1]), mod[2]) :
                make3 (AND, source[k + 1], comp (mod[1]), mod[2])),
      (k > 11 ? make4 (AND, source[15], comp (mod[0]), mod[1], mod[2]) :
                make3 (AND, source[k + 4], mod[1], mod[2])));
do4 (shift[16], OR,
    make2 (AND, comp (mod[2]), source[15]),
    make3 (AND, comp (mod[2]), mod[1], make3 (OR, source[14], source[13], source[12])),
    make3 (AND, mod[2], comp (mod[1]), source[0]),
    make3 (AND, mod[2], mod[1], source[3]));   /* "carry" */
do3 (shift[17], OR,
    make3 (AND, comp (mod[2]), comp (mod[1]), make_xor (source[15], source[14])),
    make4 (AND, comp (mod[2]), mod[1],
              make5 (OR, source[15], source[14], source[13], source[12], source[11]),
              make5 (OR, comp (source[15]), comp (source[14]), comp (source[13]),
                        comp (source[12]), comp (source[11]))),
    make3 (AND, mod[2], mod[1], make3 (OR, source[0], source[1], source[2])));   /* "overflow" */
```

This code is used in section 41.

43. RISC management. The *run_risc* procedure takes a gate graph output by *risc* and simulates its behavior, given the contents of its read-only memory. (See the demonstration program TAKE_RISC, which appears in a module by itself, for a typical illustration of how *run_risc* might be used.)

This procedure clears the simulated machine and begins executing the program that starts at address 0. It stops when it gets to an address greater than the size of read-only memory supplied. One way to stop it is therefore to execute a command such as `#0f00`, which will transfer control to location `#ffff`; even better is `#0f8f`, which transfers to location `#ffff` without changing the status of `S` and `N`. However, if the given read-only memory contains a full set of 2^{16} words, *run_risc* will never stop.

When *run_risc* does stop, it returns 0 and puts the final contents of the simulated registers into the global array *risc_state*. Or, if *g* was not a decent graph, *run_risc* returns a negative value and leaves *risc_state* untouched.

```
⟨The run_risc routine 43⟩ ≡
  long run_risc(g, rom, size, trace_regs)
      Graph *g;    /* graph output by risc */
      unsigned long rom[];    /* contents of read-only memory */
      unsigned long size;    /* length of rom vector */
      unsigned long trace_regs;    /* if nonzero, this many registers will be traced */
  { register unsigned long l;    /* memory address */
    register unsigned long m;    /* memory or register contents */
    register Vertex *v;    /* the current gate of interest */
    register Arc *a;    /* the current output list element of interest */
    register long k, r;    /* general-purpose indices */
    long x, s, n, c, o;    /* status bits */
    if (trace_regs) ⟨Print a headline 44⟩;
    r = gate_eval(g, "0", Λ);    /* reset the RISC by turning off the RUN bit */
    if (r < 0) return r;    /* not a valid gate graph! */
    g→vertices→val = 1;    /* turn the RUN bit on */
    while (1) {
      for (a = g→outs, l = 0; a; a = a→next) l = 2 * l + a→tip→val;    /* set l = memory address */
      if (trace_regs) ⟨Print register contents 46⟩;
      if (l ≥ size) break;    /* stop if memory check occurs */
      for (v = g→vertices + 1, m = rom[l]; v ≤ g→vertices + 16; v++, m >>= 1) v→val = m & 1;
          /* store bits of memory word in the input gates */
      gate_eval(g, Λ, Λ);    /* do another RISC cycle */
    }
    if (trace_regs) ⟨Print a footline 45⟩;
    ⟨Dump the register contents into risc_state 47⟩;
    return 0;
  }
```

This code is used in section 7.

44. If tracing is requested, we write on the standard output file.

```
⟨Print a headline 44⟩ ≡
  {
    for (r = 0; r < trace_regs; r++) printf("␣r%-2ld␣", r);    /* register names */
    printf("␣P␣XSNKV␣MEM\n");    /* prog, extra, status bits, memory */
  }
```

This code is used in section 43.

45. ⟨ Print a footline 45 ⟩ ≡
```
  printf("Execution␣terminated␣with␣memory␣address␣%04lx.\n", l);
```
This code is used in section 43.

46. Here we peek inside the circuit to see what values are about to be latched.

⟨ Print register contents 46 ⟩ ≡
```
  {
    for (r = 0; r < trace_regs; r++) {
      v = g→vertices + (16 * r + 47);    /* most significant bit of register r */
      m = 0;
      if (v→typ ≡ 'L')
        for (k = 0, m = 0; k < 16; k++, v--) m = 2 * m + v→alt→val;
      printf("%04lx␣", m);
    }
    for (k = 0, m = 0, v = g→vertices + 26; k < 10; k++, v--) m = 2 * m + v→alt→val;    /* prog */
    x = (g→vertices + 31)→alt→val;    /* extra */
    s = (g→vertices + 27)→alt→val;    /* sign */
    n = (g→vertices + 28)→alt→val;    /* nonzero */
    c = (g→vertices + 29)→alt→val;    /* carry */
    o = (g→vertices + 30)→alt→val;    /* overflow */
    printf("%03lx%c%c%c%c%c␣", m ≪ 2, x ? 'X' : '.', s ? 'S' : '.', n ? 'N' : '.', c ? 'K' : '.',
        o ? 'V' : '.');
    if (l ≥ size) printf("????\n");
    else printf("%04lx\n", rom[l]);
  }
```
This code is used in section 43.

47. ⟨ Dump the register contents into *risc_state* 47 ⟩ ≡
```
  for (r = 0; r < 16; r++) {
    v = g→vertices + (16 * r + 47);    /* most significant bit of register r */
    m = 0;
    if (v→typ ≡ 'L')
      for (k = 0, m = 0; k < 16; k++, v--) m = 2 * m + v→alt→val;
    risc_state[r] = m;
  }
  for (k = 0, m = 0, v = g→vertices + 26; k < 10; k++, v--) m = 2 * m + v→alt→val;    /* prog */
  m = 4 * m + (g→vertices + 31)→alt→val;    /* extra */
  m = 2 * m + (g→vertices + 27)→alt→val;    /* sign */
  m = 2 * m + (g→vertices + 28)→alt→val;    /* nonzero */
  m = 2 * m + (g→vertices + 29)→alt→val;    /* carry */
  m = 2 * m + (g→vertices + 30)→alt→val;    /* overflow */
  risc_state[16] = m;    /* program register and status bits go here */
  risc_state[17] = l;    /* this is the out-of-range address that caused termination */
```
This code is used in section 43.

48. ⟨ Global variables 48 ⟩ ≡
```
  unsigned long risc_state[18];
```
This code is used in section 7.

49. Generalized gate graphs. For intermediate computations, it is convenient to allow two additional types of gates:

'C' denotes a constant gate of value *z.I*.

'=' denotes a copy of a previous gate; utility field *alt* points to that previous gate.

Such gates might appear anywhere in the graph, possibly interspersed with the inputs and latches.

Here is a simple subroutine that prints a symbolic representation of a generalized gate graph on the standard output file:

```
#define bit z.I   /* field containing the constant value of a 'C' gate */
#define print_gates p_gates   /* abbreviation makes chopped-off name unique */
⟨ The print_gates routine 49 ⟩ ≡
  static void pr_gate(v)
      Vertex *v;
  { register Arc *a;
    printf("%s␣=␣", v→name);
    switch (v→typ) {
    case 'I': printf("input"); break;
    case 'L': printf("latch");
      if (v→alt) printf("ed␣%s", v→alt→name);
      break;
    case '~': printf("~␣"); break;
    case 'C': printf("constant␣%ld", v→bit);
      break;
    case '=': printf("copy␣of␣%s", v→alt→name);
    }
    for (a = v→arcs; a; a = a→next) {
      if (a ≠ v→arcs) printf("␣%c␣", (char) v→typ);
      printf(a→tip→name);
    }
    printf("\n");
  }
  void print_gates(g)
      Graph *g;
  { register Vertex *v;
    register Arc *a;
    for (v = g→vertices; v < g→vertices + g→n; v++) pr_gate(v);
    for (a = g→outs; a; a = a→next)
      if (is_boolean(a→tip)) printf("Output␣%ld\n", the_boolean(a→tip));
      else printf("Output␣%s\n", a→tip→name);
  }
```

This code is used in section 7.

50. ⟨ gb_gates.h 1 ⟩ +≡
```
#define bit z.I
```

51. The *reduce* routine takes a generalized graph g and uses the identities $\overline{\overline{x}} = x$ and

$$\begin{array}{llll} x \wedge 0 = 0, & x \wedge 1 = x, & x \wedge x = x, & x \wedge \overline{x} = 0, \\ x \vee 0 = x, & x \vee 1 = 1, & x \vee x = x, & x \vee \overline{x} = 1, \\ x \oplus 0 = x, & x \oplus 1 = \overline{x}, & x \oplus x = 0, & x \oplus \overline{x} = 1, \end{array}$$

to create an equivalent graph having no `'C'` or `'='` or obviously redundant gates. The reduced graph also excludes any gates that are not used directly or indirectly in the computation of the output values.

```
⟨ Internal subroutines 11 ⟩ +≡
  static Graph *reduce(g)
      Graph *g;
  { register Vertex *u, *v;    /* the current vertices of interest */
    register Arc *a, *b;    /* the current arcs of interest */
    Arc *aa, *bb;    /* their predecessors */
    Vertex *latch_ptr;    /* top of the latch list */
    long n = 0;    /* the number of marked gates */
    Graph *new_graph;    /* the reduced gate graph */
    Vertex *next_vert = Λ, *max_next_vert = Λ;    /* allocation of new vertices */
    Arc *avail_arc = Λ;    /* list of recycled arcs */
    Vertex *sentinel;    /* end of the vertices */
    if (g ≡ Λ) panic(missing_operand);    /* where is g? */
    sentinel = g→vertices + g→n;
    while (1) {
      latch_ptr = Λ;
      for (v = g→vertices; v < sentinel; v++) ⟨ Reduce gate v, if possible, or put it on the latch list 53 ⟩;
      ⟨ Check to see if any latch has become constant; if not, break 52 ⟩;
    }
    ⟨ Mark all gates that are used in some output 60 ⟩;
    ⟨ Copy all marked gates to a new graph 62 ⟩;
    gb_recycle(g);
    return new_graph;
  }
```

52. We will link latches together via their $v.V$ fields.

```
⟨ Check to see if any latch has become constant; if not, break 52 ⟩ ≡
  { char no_constants_yet = 1;
    for (v = latch_ptr; v; v = v→v.V) {
      u = v→alt;    /* the gate whose value will be latched */
      if (u→typ ≡ '=') v→alt = u→alt;
      else if (u→typ ≡ 'C') {
        v→typ = 'C'; v→bit = u→bit; no_constants_yet = 0;
      }
    }
    if (no_constants_yet) break;
  }
```

This code is used in section 51.

53. **#define** *foo* *x.V* /* link field used to find all the gates later */

⟨ Reduce gate v, if possible, or put it on the latch list 53 ⟩ ≡

```
{
  switch (v→typ) {
  case 'L': v→v.V = latch_ptr; latch_ptr = v; break;
  case 'I': case 'C': break;
  case '=': u = v→alt;
    if (u→typ ≡ '=') v→alt = u→alt;
    else if (u→typ ≡ 'C') {
      v→bit = u→bit; goto make_v_constant;
    }
    break;
  case NOT: ⟨ Try to reduce an inverter, then goto done 54 ⟩;
  case AND: ⟨ Try to reduce an AND gate 55 ⟩; goto test_single_arg;
  case OR: ⟨ Try to reduce an OR gate 56 ⟩; goto test_single_arg;
  case XOR: ⟨ Try to reduce an EXCLUSIVE-OR gate 57 ⟩;
  test_single_arg:
    if (v→arcs→next) break;
    v→alt = v→arcs→tip;
  make_v_eq: v→typ = '='; goto make_v_arcless;
  make_v_1: v→bit = 1; goto make_v_constant;
  make_v_0: v→bit = 0;
  make_v_constant: v→typ = 'C';
  make_v_arcless: v→arcs = Λ;
  }
  v→bar = Λ;     /* this field will point to the complement, if computed later */
done: v→foo = v + 1;     /* this field will link all the vertices together */
}
```

This code is used in section 51.

54. ⟨ Try to reduce an inverter, then **goto** *done* 54 ⟩ ≡

```
u = v→arcs→tip;
if (u→typ ≡ '=') u = v→arcs→tip = u→alt;
if (u→typ ≡ 'C') {
  v→bit = 1 − u→bit; goto make_v_constant;
} else if (u→bar) {     /* this inverse already computed */
  v→alt = u→bar; goto make_v_eq;
} else {
  u→bar = v; v→bar = u; goto done;
}
```

This code is used in section 53.

55. ⟨ Try to reduce an AND gate 55 ⟩ ≡

```
for ($a = v\to arcs, aa = \Lambda$; $a$; $a = a\to next$) {
  $u = a\to tip$;
  if ($u\to typ \equiv$ '=') $u = a\to tip = u\to alt$;
  if ($u\to typ \equiv$ 'C') {
    if ($u\to bit \equiv 0$) goto make_v_0;
    goto bypass_and;
  } else for ($b = v\to arcs$; $b \neq a$; $b = b\to next$) {
      if ($b\to tip \equiv u$) goto bypass_and;
      if ($b\to tip \equiv u\to bar$) goto make_v_0;
    }
  $aa = a$; continue;
bypass_and:
  if ($aa$) $aa\to next = a\to next$;
  else $v\to arcs = a\to next$;
}
if ($v\to arcs \equiv \Lambda$) goto make_v_1;
```

This code is used in section 53.

56. ⟨ Try to reduce an OR gate 56 ⟩ ≡

```
for ($a = v\to arcs, aa = \Lambda$; $a$; $a = a\to next$) {
  $u = a\to tip$;
  if ($u\to typ \equiv$ '=') $u = a\to tip = u\to alt$;
  if ($u\to typ \equiv$ 'C') {
    if ($u\to bit$) goto make_v_1;
    goto bypass_or;
  } else for ($b = v\to arcs$; $b \neq a$; $b = b\to next$) {
      if ($b\to tip \equiv u$) goto bypass_or;
      if ($b\to tip \equiv u\to bar$) goto make_v_1;
    }
  $aa = a$; continue;
bypass_or:
  if ($aa$) $aa\to next = a\to next$;
  else $v\to arcs = a\to next$;
}
if ($v\to arcs \equiv \Lambda$) goto make_v_0;
```

This code is used in section 53.

57. ⟨ Try to reduce an EXCLUSIVE-OR gate 57 ⟩ ≡

```
{ long cmp = 0;
  for (a = v→arcs, aa = Λ; a; a = a→next) {
    u = a→tip;
    if (u→typ ≡ '=') u = a→tip = u→alt;
    if (u→typ ≡ 'C') {
      if (u→bit) cmp = 1 − cmp;
      goto bypass_xor;
    } else for (bb = Λ, b = v→arcs; b ≠ a; b = b→next) {
        if (b→tip ≡ u) goto double_bypass;
        if (b→tip ≡ u→bar) {
          cmp = 1 − cmp;
          goto double_bypass;
        }
        bb = b; continue;
      double_bypass:
        if (bb) bb→next = b→next;
        else v→arcs = b→next;
        goto bypass_xor;
      }
    aa = a; continue;
  bypass_xor:
    if (aa) aa→next = a→next;
    else v→arcs = a→next;
    a→a.A = avail_arc;
    avail_arc = a;
  }
  if (v→arcs ≡ Λ) {
    v→bit = cmp;
    goto make_v_constant;
  }
  if (cmp) ⟨ Complement one argument of v 58 ⟩;
}
```

This code is used in section 53.

58. ⟨ Complement one argument of v 58 ⟩ ≡

```
{
  for (a = v→arcs; ; a = a→next) {
    u = a→tip;
    if (u→bar) break;      /* good, the complement is already known */
    if (a→next ≡ Λ) {      /* oops, this is our last chance */
      ⟨ Create a new vertex for complement of u 59 ⟩;
      break;
    }
  }
  a→tip = u→bar;
}
```

This code is used in section 57.

59. Here we've come to a subtle point: If a lot of XOR gates involve an input that is set to the constant value 1, the "reduced" graph might actually be larger than the original, in the sense of having more vertices (although fewer arcs). Therefore we must have the ability to allocate new vertices during the reduction phase of *reduce*. At least one arc has been added to the *avail_arc* list whenever we reach this portion of the program.

```
⟨ Create a new vertex for complement of u 59 ⟩ ≡
  if (next_vert ≡ max_next_vert) {
    next_vert = gb_typed_alloc(7, Vertex, g→aux_data);
    if (next_vert ≡ Λ) {
      gb_recycle(g);
      panic(no_room + 1);      /* can't get auxiliary storage! */
    }
    max_next_vert = next_vert + 7;
  }
  next_vert→typ = NOT;
  sprintf(name_buf, "%s~", u→name);
  next_vert→name = gb_save_string(name_buf);
  next_vert→arcs = avail_arc;      /* this is known to be non-Λ */
  avail_arc→tip = u;
  avail_arc = avail_arc→a.A;
  next_vert→arcs→next = Λ;
  next_vert→bar = u;
  next_vert→foo = u→foo;
  u→foo = u→bar = next_vert++;
```

This code is used in section 58.

60. During the marking phase, we will use the *w.V* field to link the list of nodes-to-be-marked. That field will turn out to be non-Λ only in the marked nodes. (We no longer use its former meaning related to complementation, so we call it *lnk* instead of *bar*.)

```
#define lnk w.V      /* stack link for marking */

⟨ Mark all gates that are used in some output 60 ⟩ ≡
  {
    for (v = g→vertices; v ≠ sentinel; v = v→foo) v→lnk = Λ;
    for (a = g→outs; a; a = a→next) {
      v = a→tip;
      if (is_boolean(v)) continue;
      if (v→typ ≡ '=') v = a→tip = v→alt;
      if (v→typ ≡ 'C') {      /* this output is constant, so make it boolean */
        a→tip = (Vertex *) v→bit;
        continue;
      }
      ⟨ Mark all gates that are used to compute v 61 ⟩;
    }
  }
```

This code is used in section 51.

61. ⟨ Mark all gates that are used to compute v 61 ⟩ ≡

```
if (v→lnk ≡ Λ) {
  v→lnk = sentinel;     /* v now represents the top of the stack of nodes to be marked */
  do {
    n++;
    b = v→arcs;
    if (v→typ ≡ 'L') {
      u = v→alt;     /* latch vertices have a "hidden" dependency */
      if (u < v) n++;     /* latched input value will get a special gate */
      if (u→lnk ≡ Λ) {
        u→lnk = v→lnk;
        v = u;
      } else v = v→lnk;
    } else v = v→lnk;
    for ( ; b; b = b→next) {
      u = b→tip;
      if (u→lnk ≡ Λ) {
        u→lnk = v;
        v = u;
      }
    }
  } while (v ≠ sentinel);
}
```

This code is used in section 60.

62. It is easier to copy a directed acyclic graph than to copy a general graph, but we do have to contend with the feedback in latches.

```
#define reverse_arc_list(alist)
          { for (aa = alist, b = Λ; aa; b = aa, aa = a) {
              a = aa→next;
              aa→next = b;
            }
            alist = b; }
```

⟨ Copy all marked gates to a new graph 62 ⟩ ≡

```
  new_graph = gb_new_graph(n);
  if (new_graph ≡ Λ) {
    gb_recycle(g);
    panic(no_room + 2);      /* out of memory */
  }
  strcpy(new_graph→id, g→id);
  strcpy(new_graph→util_types, "ZZZIIVZZZZZZZA");
  next_vert = new_graph→vertices;
  for (v = g→vertices, latch_ptr = Λ; v ≠ sentinel; v = v→foo) {
    if (v→lnk) {      /* yes, v is marked */
      u = v→lnk = next_vert++;      /* make note of where we've copied it */
      ⟨ Make u a copy of v; put it on the latch list if it's a latch 63 ⟩;
    }
  }
  ⟨ Fix up the alt fields of the newly copied latches 64 ⟩;
  reverse_arc_list(g→outs);
  for (a = g→outs; a; a = a→next) {
    b = gb_virgin_arc();
    b→tip = is_boolean(a→tip) ? a→tip : a→tip→lnk;
    b→next = new_graph→outs;
    new_graph→outs = b;
  }
```

This code is used in section 51.

63. ⟨ Make u a copy of v; put it on the latch list if it's a latch 63 ⟩ ≡

```
  u→name = gb_save_string(v→name);
  u→typ = v→typ;
  if (v→typ ≡ 'L') {
    u→alt = latch_ptr; latch_ptr = v;
  }
  reverse_arc_list(v→arcs);
  for (a = v→arcs; a; a = a→next) gb_new_arc(u, a→tip→lnk, a→len);
```

This code is used in section 62.

64. ⟨ Fix up the *alt* fields of the newly copied latches 64 ⟩ ≡

```
while (latch_ptr) {
   u = latch_ptr→lnk;      /* the copy of a latch */
   v = u→alt;
   u→alt = latch_ptr→alt→lnk;
   latch_ptr = v;
   if (u→alt < u) ⟨ Replace u→alt by a new gate that copies an input 65 ⟩;
}
```

This code is used in section 62.

65. Suppose we had a latch whose value was originally the AND of two inputs, where one of those inputs has now been set to 1. Then the latch should still refer to a subsequent gate, equal to the value of the other input on the previous cycle. We create such a gate here, making it an OR of two identical inputs. We do this because we're not supposed to leave any '=' in the result of *reduce*, and because every OR is supposed to have at least two inputs.

⟨ Replace *u→alt* by a new gate that copies an input 65 ⟩ ≡

```
{
   v = u→alt;      /* the input gate that should be copied for latching */
   u→alt = next_vert++;
   sprintf(name_buf, "%s>%s", v→name, u→name);
   u = u→alt;
   u→name = gb_save_string(name_buf);
   u→typ = OR;
   gb_new_arc(u, v, DELAY); gb_new_arc(u, v, DELAY);
}
```

This code is used in section 64.

66. Parallel multiplication. Now comes the *prod* routine, which constructs a rather different network of gates, based this time on a divide-and-conquer paradigm. Let's take a breather before we tackle it.

(Deep breath.)

The subroutine call *prod*(m, n) creates a network for the binary multiplication of unsigned m-bit numbers by n-bit numbers, assuming that $m \geq 2$ and $n \geq 2$. There is no upper limit on the sizes of m and n, except of course the limits imposed by the size of memory in which this routine is run.

The overall strategy used by *prod* is to start with a generalized gate graph for multiplication in which many of the gates are identically zero or copies of other gates. Then the *reduce* routine will perform local optimizations leading to the desired result. Since there are no latches, some of the complexities of the general *reduce* routine are avoided.

All of the AND, OR, and XOR gates of the network returned by *prod* have exactly two inputs. The depth of the circuit (i.e., the length of its longest path) is $3 \log m / \log 1.5 + \log(m+n)/\log \phi + O(1)$, where $\phi = (1+\sqrt{5})/2$ is the golden ratio. The grand total number of gates is $6mn + 5m^2 + O\bigl((m+n)\log(m+n)\bigr)$.

There is a demonstration program called MULTIPLY that uses *prod* to compute products of large integers.

```
⟨ The prod routine 66 ⟩ ≡
  Graph *prod(m, n)
      unsigned long m, n;      /* lengths of the binary numbers to be multiplied */
  { ⟨ Local variables for prod 68 ⟩
    if (m < 2) m = 2;
    if (n < 2) n = 2;
    ⟨ Allocate space for a temporary graph g and for auxiliary tables 67 ⟩;
    ⟨ Fill g with generalized gates that do parallel multiplication 70 ⟩;
    if (gb_trouble_code) {
      gb_recycle(g); panic(alloc_fault);      /* too big */
    }
    g = reduce(g);
    return g;      /* if g ≡ Λ, the panic_code was set by reduce */
  }
```

This code is used in section 7.

67. The divide-and-conquer recurrences used in this network lead to interesting patterns. First we use a method for parallel column addition that reduces the sum of three numbers to the sum of two numbers. Repeated use of this reduction makes it possible to reduce the sum of m numbers to a sum of just two numbers, with a total circuit depth that satisfies the recurrence $T(3N) = T(2N) + O(1)$. Then when the result has been reduced to a sum of two numbers, we use a parallel addition scheme based on recursively "golden sectioning the data"; in other words, the recursion partitions the data into two parts such that the ratio of the larger part to the smaller part is approximately ϕ. This technique proves to be slightly better than a binary partition would be, both asymptotically and for small values of $m + n$.

We define $\operatorname{flog} N$, the Fibonacci logarithm of N, to be the smallest nonnegative integer k such that $N \le F_{k+1}$. Let $N = m + n$. Our parallel adder for two numbers of N bits will turn out to have depth at most $2+\operatorname{flog} N$. The unreduced graph g in our circuit for multiplication will have fewer than $(6m+3\operatorname{flog} N)N$ gates.

```
⟨ Allocate space for a temporary graph g and for auxiliary tables 67 ⟩ ≡
  m_plus_n = m + n;  ⟨ Compute f = flog(m + n) 69 ⟩;
  g = gb_new_graph((6 * m − 7 + 3 * f) * m_plus_n);
  if (g ≡ Λ) panic(no_room);     /* out of memory before we're even started */
  sprintf(g→id, "prod(%lu,%lu)", m, n);
  strcpy(g→util_types, "ZZZIIVZZZZZZZA");
  long_tables = gb_typed_alloc(2 * m_plus_n + f, long, g→aux_data);
  vert_tables = gb_typed_alloc(f * m_plus_n, Vertex *, g→aux_data);
  if (gb_trouble_code) {
    gb_recycle(g);
    panic(no_room + 1);     /* out of memory trying to create auxiliary tables */
  }
```
This code is used in section 66.

68.
```
⟨ Local variables for prod 68 ⟩ ≡
  unsigned long m_plus_n;     /* guess what this variable holds */
  long f;     /* initially flog(m + n), later flog of other things */
  Graph *g;     /* graph of generalized gates, to be reduced eventually */
  long *long_tables;     /* beginning of auxiliary array of long numbers */
  Vertex **vert_tables;     /* beginning of auxiliary array of gate pointers */
```
See also sections 71 and 77.

This code is used in section 66.

69.
```
⟨ Compute f = flog(m + n) 69 ⟩ ≡
  f = 4; j = 3; k = 5;     /* j = F_f, k = F_{f+1} */
  while (k < m_plus_n) {
    k = k + j;
    j = k − j;
    f++;
  }
```
This code is used in section 67.

70. The well-known formulas for a "full adder,"

$$x + y + z = s + 2c, \qquad \text{where } s = x \oplus y \oplus z \text{ and } c = xy \vee yz \vee zx,$$

can be applied to each bit of an N-bit number, thereby providing us with a way to reduce the sum of three numbers to the sum of two.

The input gates of our network will be called x_0, x_1, …, x_{m-1}, y_0, y_1, …, y_{n-1}, and the outputs will be called z_0, z_1, …, z_{m+n-1}. The logic of the *prod* network will compute

$$(z_{m+n-1} \ldots z_1 z_0)_2 = (x_{m-1} \ldots x_1 x_0)_2 \cdot (y_{n-1} \ldots y_1 y_0)_2 \,,$$

by first considering the product to be the m-fold sum $A_0 + A_1 + \cdots + A_{m-1}$, where

$$A_j = 2^j x_j \cdot (y_{n-1} \ldots y_1 y_0)_2 \,, \qquad 0 \le j < m.$$

Then the three-to-two rule for addition is used to define further numbers A_m, A_{m+1}, …, A_{3m-5} by the scheme

$$A_{m+2j} + A_{m+2j+1} = A_{3j} + A_{3j+1} + A_{3j+2} \,, \qquad 0 \le j \le m-3.$$

[A similar but slightly less efficient scheme was used by Pratt and Stockmeyer in *Journal of Computer and System Sciences* **12** (1976), Proposition 5.3. The recurrence used here is related to the Josephus problem with step-size 3; see *Concrete Mathematics*, §3.3.] For this purpose, we compute intermediate results P_j, Q_j, and R_j by the rules

$$\begin{aligned} P_j &= A_{3j} \oplus A_{3j+1} \,; \\ Q_j &= A_{3j} \wedge A_{3j+1} \,; \\ A_{m+2j} &= P_j \oplus A_{3j+2} \,; \\ R_j &= P_j \wedge A_{3j+2} \,; \\ A_{m+2j+1} &= 2(Q_j \vee R_j) \,. \end{aligned}$$

Finally we let

$$\begin{aligned} U &= A_{3m-6} \oplus A_{3m-5} \,, \\ V &= A_{3m-6} \wedge A_{3m-5} \,; \end{aligned}$$

these are the values that would be P_{m-2} and Q_{m-2} if the previous formulas were allowed to run past $j = m-3$. The final result $Z = (z_{m+n-1} \ldots z_1 z_0)_2$ can now be expressed as

$$Z = U + 2V \,.$$

The gates of the first part of the network are conveniently obtained in groups of $N = m + n$, representing the bits of the quantities A_j, P_j, Q_j, R_j, U, and V. We will put the least significant bit of A_j in gate position *g→vertices* $+\, a(j) * N$, where $a(j) = j + 1$ for $0 \le j < m$ and $a(m + 2j + t) = m + 5j + 3 + 2t$ for $0 \le j \le m-3$, $0 \le t \le 1$.

```
⟨ Fill g with generalized gates that do parallel multiplication 70 ⟩ ≡
  next_vert = g→vertices;
  start_prefix("X"); x = first_of(m, 'I');
  start_prefix("Y"); y = first_of(n, 'I');
  ⟨ Define $A_j$ for $0 \le j < m$ 72 ⟩;
  ⟨ Define $P_j$, $Q_j$, $A_{m+2j}$, $R_j$, and $A_{m+2j+1}$ for $0 \le j \le m-3$ 73 ⟩;
  ⟨ Define $U$ and $V$ 74 ⟩;
  ⟨ Compute the final result $Z$ by parallel addition 75 ⟩;
```

This code is used in section 66.

71. ⟨ Local variables for *prod* 68 ⟩ +≡

```
register long i, j, k, l;    /* all-purpose indices */
register Vertex *v;    /* current vertex of interest */
Vertex *x, *y;    /* least-significant bits of the input gates */
Vertex *alpha, *beta;    /* least-significant bits of arguments */
```

72. ⟨ Define A_j for $0 \le j < m$ 72 ⟩ ≡

```
for (j = 0; j < m; j++) {
  numeric_prefix('A', j);
  for (k = 0; k < j; k++) {
    v = new_vert('C'); v->bit = 0;    /* this gate is the constant 0 */
  }
  for (k = 0; k < n; k++) make2(AND, x + j, y + k);
  for (k = j + n; k < m_plus_n; k++) {
    v = new_vert('C'); v->bit = 0;    /* this gate is the constant 0 */
  }
}
```

This code is used in section 70.

73. Since m is **unsigned**, it is necessary to say '$j < m - 2$' here instead of '$j \le m - 3$'.

```
#define a_pos(j) (j < m ? j + 1 : m + 5 * ((j - m) >> 1) + 3 + (((j - m) & 1) << 1))
```

⟨ Define P_j, Q_j, A_{m+2j}, R_j, and A_{m+2j+1} for $0 \le j \le m - 3$ 73 ⟩ ≡

```
for (j = 0; j < m - 2; j++) {
  alpha = g->vertices + (a_pos(3 * j) * m_plus_n);
  beta = g->vertices + (a_pos(3 * j + 1) * m_plus_n);
  numeric_prefix('P', j);
  for (k = 0; k < m_plus_n; k++) make2(XOR, alpha + k, beta + k);
  numeric_prefix('Q', j);
  for (k = 0; k < m_plus_n; k++) make2(AND, alpha + k, beta + k);
  alpha = next_vert - 2 * m_plus_n;
  beta = g->vertices + (a_pos(3 * j + 2) * m_plus_n);
  numeric_prefix('A', (long) m + 2 * j);
  for (k = 0; k < m_plus_n; k++) make2(XOR, alpha + k, beta + k);
  numeric_prefix('R', j);
  for (k = 0; k < m_plus_n; k++) make2(AND, alpha + k, beta + k);
  alpha = next_vert - 3 * m_plus_n;
  beta = next_vert - m_plus_n;
  numeric_prefix('A', (long) m + 2 * j + 1);
  v = new_vert('C'); v->bit = 0;    /* another 0, it multiplies Q ∨ R by 2 */
  for (k = 0; k < m_plus_n - 1; k++) make2(OR, alpha + k, beta + k);
}
```

This code is used in section 70.

74. Actually v_{m+n-1} will never be used (it has to be zero); but we compute it anyway. We don't have to worry about such nitty gritty details because *reduce* will get rid of all the obvious redundancy.

```
⟨ Define U and V 74 ⟩ ≡
  alpha = g→vertices + (a_pos(3 * m − 6) * m_plus_n);
  beta = g→vertices + (a_pos(3 * m − 5) * m_plus_n);
  start_prefix("U");
  for (k = 0; k < m_plus_n; k++) make2(XOR, alpha + k, beta + k);
  start_prefix("V");
  for (k = 0; k < m_plus_n; k++) make2(AND, alpha + k, beta + k);
```

This code is used in section 70.

75. Parallel addition. It's time now to take another deep breath. We have finished the parallel multiplier except for one last step, the design of a parallel adder.

The adder is based on the following theory: We want to perform the binary addition

$$\begin{array}{r} u_{N-1}\ \ldots\ u_2\ u_1\ u_0 \\ v_{N-2}\ \ldots\ v_1\ v_0 \\ \hline z_{N-1}\ \ldots\ z_2\ z_1\ z_0 \end{array}$$

where we know that $u_k + v_k \le 1$ for all k. It follows that $z_k = u_k \oplus w_k$, where $w_0 = 0$ and

$$w_k \;=\; v_{k-1} \;\lor\; u_{k-1}v_{k-2} \;\lor\; u_{k-1}u_{k-2}v_{k-3} \;\lor\; \cdots \;\lor\; u_{k-1}\ldots u_1v_0$$

for $k > 0$. The problem has therefore been reduced to the evaluation of w_1, w_2, …, w_{N-1}.

Let c_k^j denote the OR of the first j terms in the formula that defines w_k, and let d_k^j denote the j-fold product $u_{k-1}u_{k-2}\ldots u_{k-j}$. Then $w_k = c_k^k$, and we can use a recursive scheme of the form

$$c_k^j = c_k^i \lor d_k^i c_{k-i}^{j-i}\,, \qquad d_k^j = d_k^i d_{k-i}^{j-i}\,, \qquad j \ge 2,$$

to do the evaluation.

It turns out that this recursion behaves very nicely if we choose $i = \mathrm{down}[j]$, where $\mathrm{down}[j]$ is defined for $j > 1$ by the formula

$$\mathrm{down}[j] \;=\; j - F_{(\mathrm{flog}\, j)-1}\,.$$

For example, $\mathrm{flog}\,18 = 7$ because $F_7 = 13 < 18 \le 21 = F_8$, hence $\mathrm{down}[18] = 18 - F_6 = 10$.

Let us write $j \to \mathrm{down}[j]$, and consider the oriented tree on the set of all positive integers that is defined by this relation. One of the paths in this tree, for example, is $18 \to 10 \to 5 \to 3 \to 2 \to 1$. Our recurrence for $w_{18} = c_{18}^{18}$ involves c_{18}^{10}, which involves c_{18}^{5}, which involves c_{18}^{3}, and so on. In general, we will compute c_k^j for all j with $k \to^* j$, and we will compute d_k^j for all j with $k \to^+ j$. It is not difficult to prove that

$$k \;\to^*\; j \;\to\; i \qquad \text{implies} \qquad k-i \;\to^*\; j-i\,;$$

therefore the auxiliary factors c_{k-i}^{j-i} and d_{k-i}^{j-i} needed in the recurrence scheme will already have been evaluated. (Indeed, one can prove more: Let $l = \mathrm{flog}\,k$. If the complete path from k to 1 in the tree is $k = k_0 \to k_1 \to \cdots \to k_t = 1$, then the differences $k_0 - k_1$, $k_1 - k_2$, …, $k_{t-2} - k_{t-1}$ will consist of precisely the Fibonacci numbers F_{l-1}, F_{l-2}, …, F_2, except for the numbers that appear when $F_{l+1} - k$ is written as a sum of non-consecutive Fibonacci numbers.)

It can also be shown that, when $k > 1$, we have

$$\mathrm{flog}\,k = \min_{0<j<n} \max\bigl(1 + \mathrm{flog}\,j,\; 2 + \mathrm{flog}(k-j)\bigr)\,,$$

and that $\mathrm{down}[k]$ is the smallest j such that the minimum is achieved in this equation. Therefore the depth of the circuit for computing w_k from the u's and v's is exactly $\mathrm{flog}\,k$.

In particular, we can be sure that at most $3\,\mathrm{flog}\,N$ gates will be created when computing z_k, and that there will be at most $3N\,\mathrm{flog}\,N$ gates in the parallel addition portion of the circuit.

```
⟨ Compute the final result $Z$ by parallel addition 75 ⟩ ≡
   ⟨ Set up auxiliary tables to handle Fibonacci-based recurrences 76 ⟩;
   ⟨ Create the gates for $W$, remembering intermediate results that might be reused later 78 ⟩;
   ⟨ Compute the last gates $Z = U \oplus W$, and record their locations as outputs of the network 83 ⟩;
   g→n = next_vert − g→vertices;     /* reduce to the actual number of gates used */
```

This code is used in section 70.

76. After we have created a gate for w_k, we will store its address as the value of $w[k]$ in an auxiliary table. After we've created a gate for c_k^i where $i < k$ is a Fibonacci number F_{l+1} and $l = \text{flog}\, i \ge 2$, we will store its address as the value of $c[k + (l-2)N]$; the gate d_k^i will immediately follow this one. Tables of flog j and down[j] will facilitate all these manipulations.

```
⟨ Set up auxiliary tables to handle Fibonacci-based recurrences 76 ⟩ ≡
  w = vert_tables;
  c = w + m_plus_n;
  flog = long_tables;
  down = flog + m_plus_n + 1;
  anc = down + m_plus_n;
  flog[1] = 0; flog[2] = 2;
  down[1] = 0; down[2] = 1;
  for (i = 3, j = 2, k = 3, l = 3; l ≤ m_plus_n; l++) {
    if (l > k) {
      k = k + j;
      j = k − j;
      i++;     /* F_i = j < l ≤ k = F_{i+1} */
    }
    flog[l] = i;
    down[l] = l − k + j;
  }
```

This code is used in section 75.

77. ⟨ Local variables for *prod* 68 ⟩ +≡

```
  Vertex *uu, *vv;    /* pointer to u_0 and v_0 */
  Vertex **w;    /* table of pointers to w_k */
  Vertex **c;    /* table of pointers to potentially important intermediate values c_k^i */
  Vertex *cc, *dd;    /* pointers to c_k^i and d_k^i */
  long *flog;    /* table of flog values */
  long *down;    /* table of down values */
  long *anc;    /* table of ancestors of the current k */
```

78. ⟨ Create the gates for W, remembering intermediate results that might be reused later 78 ⟩ ≡

```
  vv = next_vert − m_plus_n;  uu = vv − m_plus_n;
  start_prefix("W");
  v = new_vert('C');  v→bit = 0;  w[0] = v;      /* $w_0 = 0$ */
  v = new_vert('=');  v→alt = vv;  w[1] = v;      /* $w_1 = v_0$ */
  for (k = 2;  k < m_plus_n;  k++) {
    ⟨ Set the anc table to a list of the ancestors of k in decreasing order, stopping with anc[l] = 2 79 ⟩;
    i = 1;  cc = vv + k − 1;  dd = uu + k − 1;
    while (1) {
      j = anc[l];      /* now i = down[j] */
      ⟨ Compute the gate $b_k^j = d_k^i \wedge c_{k-i}^{j-i}$ 80 ⟩;
      ⟨ Compute the gate $c_k^j = c_k^i \vee b_k^j$ 81 ⟩;
      if (flog[j] < flog[j + 1])      /* j is a Fibonacci number */
        c[k + (flog[j] − 2) ∗ m_plus_n] = v;
      if (l ≡ 0) break;
      cc = v;
      ⟨ Compute the gate $d_k^j = d_k^i \wedge d_{k-i}^{j-i}$ 82 ⟩;
      dd = v;
      i = j;
      l−−;
    }
    w[k] = v;
  }
```

This code is used in section 75.

79. If $k \to j$, we call j an "ancestor" of k because we are thinking of the tree defined by '$\to$'; this tree is rooted at $2 \to 1$.

⟨ Set the *anc* table to a list of the ancestors of k in decreasing order, stopping with $anc[l] = 2$ 79 ⟩ ≡

```
  for (l = 0, j = k;  ;  l++, j = down[j]) {
    anc[l] = j;
    if (j ≡ 2) break;
  }
```

This code is used in section 78.

80.

```
#define spec_gate(v, a, k, j, t)  v = next_vert++;
        sprintf(name_buf, "%c%ld:%ld", a, k, j);
        v→name = gb_save_string(name_buf);
        v→typ = t;
```

⟨ Compute the gate $b_k^j = d_k^i \wedge c_{k-i}^{j-i}$ 80 ⟩ ≡

```
  spec_gate(v, 'B', k, j, AND);
  gb_new_arc(v, dd, DELAY);      /* first argument is $d_k^i$ */
  f = flog[j − i];      /* get ready to compute the second argument, $c_{k-i}^{j-i}$ */
  gb_new_arc(v, f > 0 ? c[k − i + (f − 2) ∗ m_plus_n] : vv + k − i − 1, DELAY);
```

This code is used in section 78.

81. ⟨ Compute the gate $c_k^j = c_k^i \vee b_k^j$ 81 ⟩ ≡

```
if (l) {
  spec_gate(v, 'C', k, j, OR);
} else v = new_vert(OR);    /* if l is zero, this gate is $c_k^k = w_k$ */
gb_new_arc(v, cc, DELAY);    /* first argument is $c_k^i$ */
gb_new_arc(v, next_vert − 2, DELAY);    /* second argument is $b_k^j$ */
```

This code is used in section 78.

82. Here we reuse the value $f = \mathrm{flog}(j - i)$ computed a minute ago.

⟨ Compute the gate $d_k^j = d_k^i \wedge d_{k-i}^{j-i}$ 82 ⟩ ≡

```
spec_gate(v, 'D', k, j, AND);
gb_new_arc(v, dd, DELAY);    /* first argument is $d_k^i$ */
gb_new_arc(v, f > 0 ? c[k − i + (f − 2) * m_plus_n] + 1 : uu + k − i − 1, DELAY);    /* $d_{k-i}^{j-i}$ */
```

This code is used in section 78.

83. The output list will contain the gates in "big-endian order" z_{m+n-1}, ..., z_1, z_0, because we insert them into the *outs* list in little-endian order.

⟨ Compute the last gates $Z = U \oplus W$, and record their locations as outputs of the network 83 ⟩ ≡

```
start_prefix("Z");
for (k = 0; k < m_plus_n; k++) { register Arc *a = gb_virgin_arc();
  a→tip = make2(XOR, uu + k, w[k]);
  a→next = g→outs;
  g→outs = a;
}
```

This code is used in section 75.

84. Partial evaluation. The subroutine call *partial_gates*$(g, r, prob, seed, buf)$ creates a new gate graph from a given gate graph g by "partial evaluation," i.e., by setting some of the inputs to constant values and simplifying the result. The new graph is usually smaller than g; it might, in fact, be a great deal smaller. Graph g is destroyed in the process.

The first r inputs of g are retained unconditionally. Each remaining input is retained with probability $prob/65536$, and if not retained it is assigned a random constant value. For example, about half of the inputs will become constant if $prob = 32768$. The *seed* parameter defines a machine-independent source of random numbers, and it may be given any value between 0 and $2^{31} - 1$.

If the *buf* parameter is non-null, it should be the address of a string. In such a case, *partial_gates* will put a record of its partial evaluation into that string; *buf* will contain one character for each input gate after the first r, namely `'*'` if the input was retained, `'0'` if it was set to 0, or `'1'` if it was set to 1.

The new graph will contain only gates that contribute to the computation of at least one output value. Therefore some input gates might disappear even though they were supposedly "retained," i.e., even though their value has not been set constant. The *name* field of a vertex can be used to determine exactly which input gates have survived.

If graph g was created by *risc*, users will probably want to make $r \geq 1$, since the whole RISC circuit collapses to zero whenever its first input '`RUN`' is set to 0.

An interesting class of graphs is produced by the function call *partial_gates*$(prod(m, n), m, 0, seed, \Lambda)$, which creates a graph corresponding to a circuit that multiplies a given m-bit number by a fixed (but randomly selected) n-bit constant. If the constant is not zero, all m of the "retained" input gates necessarily survive. The demo program called MULTIPLY illustrates such circuits.

The graph g might be a generalized network; that is, it might involve the `'C'` or `'='` gates described earlier. Notice that if r is sufficiently large, *partial_gates* becomes equivalent to the *reduce* routine. Therefore we need not make that private routine public.

As usual, the result will be Λ, and *panic_code* will be set, if *partial_gates* is unable to complete its task.

```
⟨ The partial_gates routine 84 ⟩ ≡
  Graph *partial_gates(g, r, prob, seed, buf)
      Graph *g;     /* generalized gate graph */
      unsigned long r;     /* the number of initial gates to leave untouched */
      unsigned long prob;     /* scaled probability of not touching subsequent input gates */
      long seed;     /* seed value for random number generation */
      char *buf;     /* optional parameter for information about partial assignment */
  { register Vertex *v;     /* the current gate of interest */
    if (g ≡ Λ) panic(missing_operand);     /* where is g? */
    gb_init_rand(seed);     /* get them random numbers rolling */
    for (v = g→vertices + r; v < g→vertices + g→n; v++)
      switch (v→typ) {
      case 'C': case '=': continue;     /* input gates might still follow */
      case 'I':
        if ((gb_next_rand() ≫ 15) ≥ prob) {
          v→typ = 'C'; v→bit = gb_next_rand() ≫ 30;
          if (buf) *buf++ = v→bit + '0';
        } else if (buf) *buf++ = '*';
        break;
      default: goto done;     /* no more input gates can follow */
      }
  done:
    if (buf) *buf = 0;     /* terminate the string */
    g = reduce(g);
    ⟨ Give the reduced graph a suitable id 85 ⟩;
    return g;     /* if (g ≡ Λ), a panic_code has been set by reduce */
  }
```

This code is used in section 7.

85. The *buf* parameter is not recorded in the graph's *id* field, since it has no effect on the graph itself.

⟨ Give the reduced graph a suitable *id* 85 ⟩ ≡

```
if (g) {
  strcpy(name_buf, g→id);
  if (strlen(name_buf) > 54) strcpy(name_buf + 51, "...");
  sprintf(g→id, "partial_gates(%s,%lu,%lu,%ld)", name_buf, r, prob, seed);
}
```

This code is used in section 84.

86. Index. Here is a list that shows where the identifiers of this program are defined and used.

a: 3, 36, 43, 49, 51, 83.
a_pos: 73, 74.
aa: 51, 55, 56, 57, 62.
add: 38.
alist: 62.
alloc_fault: 8, 66.
alpha: 71, 73, 74.
alt: 2, 3, 32, 46, 47, 49, 52, 53, 54, 55, 56, 57, 60, 61, 63, 64, 65, 78.
anc: 76, 77, 78, 79.
AND: 2, 6, 15, 21, 22, 23, 24, 26, 27, 30, 31, 32, 34, 35, 36, 38, 39, 41, 42, 53, 66, 72, 73, 74, 80, 82.
arcs: 6, 49, 53, 54, 55, 56, 57, 58, 59, 61, 63.
aux_data: 59, 67.
avail_arc: 51, 57, 59.
b: 51.
bar: 14, 53, 54, 55, 56, 57, 58, 59, 60.
bb: 51, 57.
beta: 71, 73, 74.
bit: 49, 50, 52, 53, 54, 55, 56, 57, 60, 72, 73, 78, 84.
buf: 84, 85.
bypass_and: 55.
bypass_or: 56.
bypass_xor: 57.
c: 43, 77.
carry: 18, 19, 27, 35, 38, 41, 46, 47.
cc: 77, 78, 81.
change: 27, 28, 31, 32, 34, 36.
cmp: 57.
comp: 14, 15, 21, 26, 27, 30, 31, 32, 34, 35, 36, 38, 39, 41, 42.
cond: 20, 21, 27, 31, 32, 33, 35, 36.
count: 11, 12, 19.
dd: 77, 78, 80, 82.
DELAY: 13, 23, 24, 65, 80, 81, 82.
dest: 20, 21, 23, 32.
dest_match: 23, 25, 34.
diff: 25, 31, 41.
dir: 20, 21, 22.
done: 53, 54, 84.
double_bypass: 57.
down: 39, 40, 76, 77, 79.
do2: 20, 21, 31, 32, 34, 36, 41.
do3: 20, 21, 27, 31, 38, 39, 42.
do4: 20, 22, 23, 26, 27, 35, 36, 38, 42.
do5: 20, 24, 31, 35.
even_comp: 14, 23, 24, 38.
extra: 18, 19, 21, 22, 32, 44, 46, 47.
f: 68.
Fibonacci, Leonardo, numbers: 67.
first_of: 19, 70.
flog: 76, 77, 78, 80.
foo: 53, 59, 60, 62.
g: 3, 43, 49, 51, 68, 84.
gate_eval: 1, 3, 10, 43.
gb_init_rand: 84.
gb_new_arc: 13, 14, 23, 24, 63, 65, 80, 81, 82.
gb_new_graph: 16, 62, 67.
gb_next_rand: 84.
gb_recycle: 8, 51, 59, 62, 66, 67.
gb_save_string: 11, 14, 59, 63, 65, 80.
gb_trouble_code: 8, 66, 67.
gb_typed_alloc: 59, 67.
gb_virgin_arc: 36, 62, 83.
hop: 36, 37.
i: 71.
id: 16, 62, 67, 85.
imm: 20, 21, 22.
impossible: 17.
in_vec: 3, 4.
inc_dest: 22, 25, 39, 40.
ind: 20, 21, 32, 34, 35, 36.
is_boolean: 2, 49, 60, 62.
j: 71.
Josephus, Flavius, problem: 70.
jump: 31, 33, 35, 36.
k: 9, 38, 43, 71.
l: 43, 71.
latch: 32.
latch_ptr: 51, 52, 53, 62, 63, 64.
latchit: 32, 34, 35, 36.
len: 63.
lnk: 60, 61, 62, 63, 64.
log: 25, 26, 31.
long_tables: 67, 68, 76.
m: 43, 66.
m_plus_n: 67, 68, 69, 72, 73, 74, 76, 78, 80, 82, 83.
make_adder: 38, 39, 41.
make_v_arcless: 53.
make_v_constant: 53, 54, 57.
make_v_eq: 53, 54.
make_v_0: 53, 55, 56.
make_v_1: 53, 55, 56.
make_xor: 15, 30, 38, 42.
make2: 13, 15, 20, 21, 22, 23, 27, 30, 31, 32, 34, 35, 36, 38, 39, 41, 42, 72, 73, 74, 83.
make3: 13, 20, 26, 27, 31, 36, 38, 39, 41, 42.
make4: 13, 20, 32, 35, 36, 42.
make5: 13, 20, 31, 35, 36, 42.
max_next_vert: 51, 59.
mem: 18, 19, 21, 22, 24, 32, 39.
missing_operand: 51, 84.

mod: 20, 21, 26, 27, 31, 35, 41, 42.
n: 38, 43, 51, 66.
name: 11, 14, 49, 59, 63, 65, 80, 84.
name_buf: 11, 12, 14, 59, 65, 80, 85.
new_graph: 8, 9, 16, 17, 36, 51, 62.
new_vert: 11, 13, 19, 23, 24, 72, 73, 78, 81.
next: 5, 6, 36, 43, 49, 53, 55, 56, 57, 58, 59, 60, 61, 62, 63, 83.
next_loc: 25, 30, 31, 36.
next_next_loc: 25, 30, 32, 36.
next_vert: 11, 12, 14, 16, 17, 51, 59, 62, 65, 70, 73, 75, 78, 80, 81.
nextra: 32, 33.
no_constants_yet: 52.
no_room: 16, 59, 62, 67.
nonzero: 18, 19, 27, 35, 46, 47.
normal: 36, 37.
`NOT`: 2, 6, 14, 53, 59.
numeric_prefix: 12, 19, 72, 73.
nzd: 32, 33, 36.
nzs: 32, 33, 36.
o: 43.
old_dest: 23, 25, 26, 39, 41.
old_src: 22, 23, 24, 25, 36.
op: 20, 21, 27, 31, 35.
`OR`: 2, 6, 15, 21, 22, 23, 24, 26, 27, 31, 32, 34, 35, 36, 38, 39, 41, 42, 53, 65, 66, 73, 81.
out_vec: 3, 5.
outs: 2, 5, 36, 43, 49, 60, 62, 83.
overflow: 18, 19, 27, 35, 46, 47.
p_gates: 1, 49.
panic: 8, 16, 17, 51, 59, 62, 66, 67, 84.
panic_code: 8, 66, 84.
partial_gates: 1, 84.
pr_gate: 49.
Pratt, Vaughan Ronald: 70.
prefix: 11, 12, 19.
print_gates: 1, 49.
printf: 44, 45, 46, 49.
prob: 84, 85.
prod: 1, 2, 38, 66, 70, 84.
prog: 18, 19, 21, 32, 44, 46, 47.
r: 9, 43, 84.
reduce: 51, 59, 65, 66, 74, 84.
reg: 18, 19, 23, 24, 30, 34, 36.
regs: 8, 16, 19, 23, 24, 34.
rel: 20, 21, 22.
result: 20, 25, 31, 34, 35, 36.
reverse_arc_list: 62, 63.
risc: 1, 2, 8, 9, 38, 43, 84.
risc_state: 1, 43, 47, 48.
rom: 43, 46.
run_bit: 18, 19, 32, 36.
run_risc: 1, 43.
s: 43.
seed: 84, 85.
sentinel: 51, 60, 61, 62.
shift: 25, 31, 42.
sign: 18, 19, 27, 35, 46, 47.
size: 43, 46.
skip: 36, 37.
source: 22, 25, 26, 31, 36, 41, 42.
spec_gate: 80, 81, 82.
special: 36, 37.
sprintf: 11, 12, 14, 16, 59, 65, 67, 80, 85.
start_prefix: 12, 19, 21, 22, 26, 27, 29, 41, 70, 74, 78, 83.
Stockmeyer, Larry Joseph: 70.
strcpy: 12, 16, 19, 62, 67, 85.
strlen: 85.
sum: 25, 31, 41.
t: 3, 11, 13.
test_single_arg: 53.
the_boolean: 2, 49.
tip: 2, 5, 6, 36, 43, 49, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 83.
tip_value: 2, 5.
tmp: 20, 23, 24, 27.
trace_regs: 43, 44, 46.
typ: 2, 3, 11, 14, 46, 47, 49, 52, 53, 54, 55, 56, 57, 59, 60, 61, 63, 65, 80, 84.
t1: 15, 20, 38.
t2: 15, 20, 38.
t3: 20, 34, 38.
t4: 20, 34, 36, 38.
t5: 20, 30, 34, 35, 36.
u: 14, 15, 51.
up: 39, 40.
util_types: 16, 62, 67.
uu: 77, 78, 82, 83.
v: 3, 11, 13, 14, 15, 43, 49, 51, 71, 84.
val: 2, 3, 4, 6, 43, 46, 47.
vert_tables: 67, 68, 76.
vertices: 3, 4, 16, 17, 43, 46, 47, 49, 51, 60, 62, 70, 73, 74, 75, 84.
vv: 77, 78, 80.
v1: 13, 20.
v2: 13, 20.
v3: 13, 20.
v4: 13, 20.
v5: 13, 20.
w: 77.
x: 38, 43, 71.
`XOR`: 2, 6, 53, 59, 66, 73, 74, 83.

y: <u>38</u>, <u>71</u>.
z: <u>38</u>.
zz: 2.

⟨ Add the RISC data to *new_graph* 17 ⟩ Used in section 8.
⟨ Allocate space for a temporary graph g and for auxiliary tables 67 ⟩ Used in section 66.
⟨ Check to see if any latch has become constant; if not, **break** 52 ⟩ Used in section 51.
⟨ Complement one argument of v 58 ⟩ Used in section 57.
⟨ Compute $f = \mathrm{flog}(m + n)$ 69 ⟩ Used in section 67.
⟨ Compute the final result Z by parallel addition 75 ⟩ Used in section 70.
⟨ Compute the gate $b_k^j = d_k^i \wedge c_{k-i}^{j-i}$ 80 ⟩ Used in section 78.
⟨ Compute the gate $c_k^j = c_k^i \vee b_k^j$ 81 ⟩ Used in section 78.
⟨ Compute the gate $d_k^j = d_k^i \wedge d_{k-i}^{j-i}$ 82 ⟩ Used in section 78.
⟨ Compute the last gates $Z = U \oplus W$, and record their locations as outputs of the network 83 ⟩ Used in section 75.
⟨ Compute the value t of a classical logic gate 6 ⟩ Used in section 3.
⟨ Copy all marked gates to a new graph 62 ⟩ Used in section 51.
⟨ Create a new vertex for complement of u 59 ⟩ Used in section 58.
⟨ Create gates for fetching the source value 22 ⟩ Used in section 17.
⟨ Create gates for instruction decoding 21 ⟩ Used in section 17.
⟨ Create gates for the arithmetic operations 41 ⟩ Used in section 17.
⟨ Create gates for the conditional load operations 27 ⟩ Used in section 17.
⟨ Create gates for the general logic operation 26 ⟩ Used in section 17.
⟨ Create gates for the new values of S, N, K, and V 35 ⟩ Used in section 29.
⟨ Create gates for the new values of register 0 and the memory address register 36 ⟩ Used in section 29.
⟨ Create gates for the new values of registers 1 to *regs* 34 ⟩ Used in section 29.
⟨ Create gates for the new values of the program register and *extra* 32 ⟩ Used in section 29.
⟨ Create gates for the shift operations 42 ⟩ Used in section 41.
⟨ Create gates for the *next_loc* and *next_next_loc* bits 30 ⟩ Used in section 29.
⟨ Create gates for the *result* bits 31 ⟩ Used in section 29.
⟨ Create gates that bring everything together properly 29 ⟩ Used in section 17.
⟨ Create the gates for W, remembering intermediate results that might be reused later 78 ⟩ Used in section 75.
⟨ Create the inputs and latches 19 ⟩ Used in section 17.
⟨ Define A_j for $0 \le j < m$ 72 ⟩ Used in section 70.
⟨ Define P_j, Q_j, A_{m+2j}, R_j, and A_{m+2j+1} for $0 \le j \le m - 3$ 73 ⟩ Used in section 70.
⟨ Define U and V 74 ⟩ Used in section 70.
⟨ Dump the register contents into *risc_state* 47 ⟩ Used in section 43.
⟨ Fill g with generalized gates that do parallel multiplication 70 ⟩ Used in section 66.
⟨ Fix up the *alt* fields of the newly copied latches 64 ⟩ Used in section 62.
⟨ Give the reduced graph a suitable *id* 85 ⟩ Used in section 84.
⟨ Global variables 48 ⟩ Used in section 7.
⟨ Initialize *new_graph* to an empty graph of the appropriate size 16 ⟩ Used in section 8.
⟨ Internal subroutines 11, 13, 14, 15, 38, 51 ⟩ Used in section 7.
⟨ Local variables for *prod* 68, 71, 77 ⟩ Used in section 66.
⟨ Local variables for *risc* 9, 18, 20, 25, 28, 33, 37, 40 ⟩ Used in section 8.
⟨ Make u a copy of v; put it on the latch list if it's a latch 63 ⟩ Used in section 62.
⟨ Mark all gates that are used in some output 60 ⟩ Used in section 51.
⟨ Mark all gates that are used to compute v 61 ⟩ Used in section 60.
⟨ Print a footline 45 ⟩ Used in section 43.
⟨ Print a headline 44 ⟩ Used in section 43.
⟨ Print register contents 46 ⟩ Used in section 43.
⟨ Private variables 12 ⟩ Used in section 7.
⟨ Read a sequence of input values from *in_vec* 4 ⟩ Used in section 3.
⟨ Reduce gate v, if possible, or put it on the latch list 53 ⟩ Used in section 51.
⟨ Replace *u→alt* by a new gate that copies an input 65 ⟩ Used in section 64.

⟨ Set the *anc* table to a list of the ancestors of k in decreasing order, stopping with $anc[l] = 2$ 79 ⟩ Used in section 78.
⟨ Set up auxiliary tables to handle Fibonacci-based recurrences 76 ⟩ Used in section 75.
⟨ Set *inc_dest* to *old_dest* plus `SRC` 39 ⟩ Used in section 22.
⟨ Set *old_dest* to the present value of the destination register 23 ⟩ Used in section 22.
⟨ Set *old_src* to the present value of the source register 24 ⟩ Used in section 22.
⟨ Store the sequence of output values in *out_vec* 5 ⟩ Used in section 3.
⟨ The *gate_eval* routine 3 ⟩ Used in section 7.
⟨ The *partial_gates* routine 84 ⟩ Used in section 7.
⟨ The *print_gates* routine 49 ⟩ Used in section 7.
⟨ The *prod* routine 66 ⟩ Used in section 7.
⟨ The *risc* routine 8 ⟩ Used in section 7.
⟨ The *run_risc* routine 43 ⟩ Used in section 7.
⟨ Try to reduce an inverter, then **goto** *done* 54 ⟩ Used in section 53.
⟨ Try to reduce an AND gate 55 ⟩ Used in section 53.
⟨ Try to reduce an EXCLUSIVE-OR gate 57 ⟩ Used in section 53.
⟨ Try to reduce an OR gate 56 ⟩ Used in section 53.
⟨ `gb_gates.h` 1, 2, 50 ⟩

January 9, 2001 at 12:11

GB_GATES

	Section	Page
Introduction	1	1
The RISC netlist	8	5
Serial addition	38	18
RISC management	43	21
Generalized gate graphs	49	23
Parallel multiplication	66	32
Parallel addition	75	37
Partial evaluation	84	41
Index	86	43

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