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

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

risc_elte, a routine that creates a directed acyclic graph based on the logic of a simple RISC computer;
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;
run_risc_elte, a routine that can be used to play with the output of *risc_elte*.

Examples of the use of these routines can be found in the demo program TAKE_RISC.

```
⟨gb_gates_elte.h 1⟩ ≡
#define print_gates p_gates   /* abbreviation for Procrustean linkers */
  extern Graph *risc_elte();   /* make a network for a microprocessor */
  extern void print_gates();   /* write a network to standard output file */
  extern long gate_eval();   /* evaluate a network */
  extern long run_risc_elte();   /* simulate the microprocessor */
  extern unsigned long risc_state[];   /* the output of run_risc_elte */
```

See also sections 2 and 53.

2. The directed acyclic graphs produced by GB_GATES_ELTE 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_elte.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 ($a = v\to arcs$; $a$; $a = a\to next$) $t$ &= $a\to tip\to val$;
  break;
case OR: $t = 0$;
  for ($a = v\to arcs$; $a$; $a = a\to next$) $t$ |= $a\to tip\to val$;
  break;
case XOR: $t = 0$;
  for ($a = v\to arcs$; $a$; $a = a\to next$) $t \oplus= a\to tip\to val$;
  break;
case NOT: $t = 1 - v\to arcs\to tip\to val$;
  break;
```

This code is used in section 3.

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

```
#include "gb_graph.h"      /* and we will use the GB_GRAPH data structures */
  ⟨ Preprocessor definitions ⟩
  ⟨ Private variables 12 ⟩
  ⟨ Global variables 51 ⟩
  ⟨ Internal subroutines 11 ⟩
  ⟨ The gate_eval routine 3 ⟩
  ⟨ The print_gates routine 52 ⟩
  ⟨ The risc_elte routine 8 ⟩
  ⟨ The run_risc_elte routine 46 ⟩
```

8. The RISC ELTE netlist. The subroutine call *risc_elte*(*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_elte* cannot do its thing, it returns Λ (NULL) and sets *panic_code* to indicate the problem. Otherwise *risc_elte* returns a pointer to the graph.

```
#define panic(c) { panic_code = c; gb_trouble_code = 0; return Λ; }
⟨ The risc_elte routine 8 ⟩ ≡
  Graph *risc_elte(regs)
      unsigned long regs;    /* number of registers supported */
  { ⟨ Local variables for risc_elte 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_elte* 9 ⟩ ≡

```
  Graph *new_graph;    /* the graph constructed by risc */
  register long i, j, k, r;    /* all-purpose indices */
```

See also sections 18, 20, 23, 30, 37, and 39.

This code is used in section 8.

10. This RISC ELTE machine works with 16-bit registers and 16-bit data words. It assumes the existence of an external memory. The circuit has 19 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. The second input bit is the interrupt bit `INT`; it may cause an interupt if this is enabled, see below. The third input bit is the page bit `PAGE` which is always 0 by smaller systems; it cause an interrupt anyway, even during an instruction, see below. Input bits for the memory appear in "little-endian order," that is, least significant bit first. The circuit has 54 outputs. The first and second is the (new) value of the translation flag `T` and the (new) value of the extra flag `X`; they can be ignored by small systems, see below. The third is data bit `D`: it can be considered as an extension to the next 16 representing the 16 bits of a memory address register. (The output bits for the memory address register appear in "big-endian order," most significant bit first.) A small von Neumann architecture can ignore `D` and use a 64 kk word memory, other systems may use separate program and data memory (or mix the two). The remaining 35 output for write. The next output is the write bit `W`; if it is zero, no write data and the last 34 output is garbage. Otherwise, comes the (old) value of the `X` flag, the data bit `D` belonging to the 16 bit write address and 16 bit data to write; they are also in "big-endian order".

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 is called MXOR and well-known from MMIX. This sets S and N but does not change K and V.

The operations for OP = 1 and MOD = 13 is the write operation and will write into memory. The content of the destination register specified by the DST field is written to the address which is the "source" value specified by the SRC and A fields as given the general rules with two exceptions: in the case A = 0 the address is the SRC value but without sign extension and in the case A = 1 the address is the pre-decremented content of the SRC register, i. e., this is a PUSH instruction. The only case when write does not goes to data memory a PUSH instruction with SRC = 0; this may be useful in a debugger. Flags are not affected.

The operations for OP = 1 and MOD = 14 and 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 ELTE 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.

The JUMP 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.

We have now discussed everything about the machine except the operations in the case OP = 1 and MOD = 14. In the case A = 0 a LOAD operation is done: data from the address given by SRC will be loaded into the register named by DST (no flags affected). In the case A = 2 a SADD operation is done: the content of the register named by `DST` is replaced by number of one's in the register named by `SRC`; only flags `S` and `N` are affected. In the case `A` $= 3$ data from the address given by the register `SRC` will be loaded into the register named by `DST` and the content of register `SRC` is post-incremented if `SRC` $\neq$ `DST`; i. e., a `POP` instruction is done. Finally, the case `A` $= 1$ is connected to interrupts, pageing and system calls. It is called `TRAP` if `DST` $\neq 0$ and `RESUME` if `DST` $= 0$. The processor has two extra flags, the translate flag `T` and the enable flag `E`. For the normal user, both set to 1. The `TRAP` instruction set both to 0, and register 0 will set to $32 + 2 *$ `SRC`. Furthermore, it also sets register `DST` to the location of the instruction following the `TRAP`. Assembly language programmers will recognize this as a convenient way to a system call. The `RESUME` instruction make possible the return from system calls: it sets register 0 to the content of register `SRC` if `SRC` $\neq 0$, and to the content of the memory at address 0 if `SRC` $= 0$, and, finally sets flags `T` and `E` to 1. The case `SRC` $= 0$ is useful if we want to return from an interrupt. There is one more way to set `E` to 1: the `NOPE` logical instruction with `OP` $= 0$, `MOD` $= 1010$, `A` $= 1$, `SRC` $= 0$ and `DST` $\neq 0$ has this side effect (but other `NOP` instructions, for example with `OP` $= 0$, `MOD` $= 1010$, `A` $= 2$ and `DST` $=$ `SRC` $\neq 0$ has not). If the input `INT` $= 1$ and also `E` $= 1$, no new instruction is started, but `E` and `T` set to 0, the content of register 0 is written to (data) memory address 0 and register 0 is set to 1. If the input `PAGE` $= 1$, it has the same effect, but instruction not finished; operating system and auxiliar hardware together may figure out where to `RESUME` and what to correct in register content before resume.

Auxiliary hardware may simulate auxiliary registers. For example, by a system running under Unix the highest address of (data) memory may represent an interupt register, and the second highest may represent an interupt mask register. They can be read and write by the processor and by the auxiliar hardware, too. If their "and" gives a nonzero result, an `INT` is generated by the auxiliar hardware. Third highest address may contain a translation register with least significant bits `r`, `w` and `x`. If, for example, `x` $= 1$ or `T` $= 1$, then by any reading from program memory a translation is done: the most significant bits (for example, `D` plus 3 bits) of the memory address used as a relative address to the translation table. The physical address given by the most significant 13 bits of the translation table data plus the 13 offset bits of the logical address. (If no translation, then the most significant 9 bits of the physical address are zero.) The least significant 3 bits `r`, `w` and `x` of the translation table data used as protection bits. Most significant 13 bits of translation register can be used for other purposes, for example setting a bit may swich of a given `ROM` or `RAM` interval. (The translation register should be 0 by starting.)

The auxiliary hardware can help for the operating system by paging error for example on the following way: by page error during write from the least significant three bit `r`, `w` and `x` of interrupt register the bit `w` set to 1, bit `r` is set from the value of old extra flag `X` meaning extra reading and `x` is set to the complement of `D` meaning program memory writing. By any other page error `w` is set to 0, `r` is set from the new value of `X` meaning extra reading and `x` is set to the complement of `D` meaning program memory reading, i. e., execution. If no page error, they are not changed. Operating system may figure out what to do; for example if $w = 1$ then in the case $r = 0$ the instruction cousing page error is on the previous adress, in the the case $r = 1$ before it; etc.

Example programs can be found in the TAKE_RISC_ELTE 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, and 40.

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_elte(%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 output write address and flags; ELTE 27 ⟩;
⟨ Create gates for the general logic operation 28 ⟩;
⟨ Create gates for the conditional load operations 29 ⟩;
⟨ Create gates for the matrix operation; ELTE 43 ⟩;
⟨ Create gates for the arithmetic operations 42 ⟩;
⟨ Create gates for the sideway add operation; ELTE 44 ⟩;
⟨ Create gates that bring everything together properly 31 ⟩;
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_elte* 9 ⟩ +≡
```
Vertex *run_bit;   /* the RUN input */
Vertex *int_bit;   /* the INT input;ELTE */
Vertex *page_bit;   /* the PAGE input; ELTE */
Vertex *mem[16];   /* 16 bits of input from 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 X: are we doing an extra memory cycle? */
Vertex *enabled;   /* latched status bit E: is interupt enabled? ELTE */
Vertex *translate;   /* latched status bit T: must translate? ELTE */
Vertex *xresume;   /* latched status bit XR: extra cycle for resume? ELTE */
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');
strcpy(prefix, "INT"); count = −1; interupt_bit = new_vert('I');      /* ELTE */
strcpy(prefix, "PAGE"); count = −1; page_bit = new_vert('I');      /* ELTE */
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');
strcpy(prefix, "E"); enable = new_vert('L');      /* ELTE */
strcpy(prefix, "T"); translate = new_vert('L');      /* ELTE */
strcpy(prefix, "XR"); xresume = new_vert('L');      /* ELTE */
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_elte* 9 ⟩ +≡

```
  Vertex *t1, *t2, *t3, *t4, *t5, *t6, *t7, *t8;
    /* temporary holds to force evaluation order; ELTE */
  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 */
  Vertex *dirind;   /* dir or ind? ELTE */
  Vertex *jump;   /* is this command a JUMP, assuming cond is false? ELTE moved */
  Vertex *write;   /* is this command a WRITE, assuming cond is false? ELTE */
  Vertex *mixed;   /* is this command a MIXED, assuming cond is false? ELTE */
  Vertex *nochangeflags;   /* should the flags changed? ELTE */
  Vertex *brk;   /* enabled interupt or page error? ELTE */
  Vertex *nzs;   /* is the SRC field nonzero? ELTE moved */
  Vertex *push;   /* is this command a PUSH? ELTE */
  Vertex *pop;   /* is this command a POP? ELTE */
  Vertex *trap;   /* is this command a TRAP? ELTE */
  Vertex *resume0;   /* is this command a RESUME with SRC = 0? ELTE */
  Vertex *resume;   /* is this command an other RESUME? ELTE */
  Vertex *load_imm;   /* is this command an immediate LOAD? ELTE */
  Vertex *nope;   /* is this command a NOPE, assuming cond = 0, etc.? ELTE */
```

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

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

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]));
}
dirind = make2(AND, comp(extra), mem[5]);   /* A ≥ 2; ELTE */
jump = make5(AND, op, mod[0], mod[1], mod[2], mod[3]);   /* assume cond = 0; ELTE moved */
mixed = make5(AND, op, comp(mod[0]), mod[1], mod[2], mod[3]);   /* assume cond = 0; ELTE */
write = make5(AND, op, mod[0], comp(mod[1]), mod[2], mod[3]);   /* assume cond = 0; ELTE */
do2(nochangeflags, OR, cond, ind, jump, write, make2(AND, mixed, comp(cond), dir));   /* ELTE */
do2(brk, OR, page_bit, make3(AND, comp(extra), int_bit, enabled));   /* ELTE */
nzs = make4(OR, mem[0], mem[1], mem[2], mem[3]);   /* ELTE moved */
nzd = make4(OR, dest[0], dest[1], dest[2], dest[3]);   /* ELTE moved */
push = make3(AND, comp(cond), write, rel);   /* ELTE */
pop = make3(AND, comp(cond), mixed, ind);   /* ELTE */
trap = make4(AND, comp(cond), mixed, rel, nzd);   /* ELTE */
resume0 = make5(AND, comp(cond), mixed, rel, comp(nzd), comp(nzs));
  /* resume with zero source; ELTE */
resume = make5(AND, comp(cond), mixed, rel, comp(nzd), nzs);   /* other resume; ELTE */
load_imm = make3(AND, comp(cond), mixed, imm);   /* ELTE */
nope = make5(AND, comp(op), mod[0], comp(mod[1]), mod[2], comp(mod[3]));
  /* assume cond = 0, etc; ELTE */
```

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 24 ⟩;
⟨ Set old_src to the present value of the source register 25 ⟩;
⟨ Set inc_dest to old_dest plus SRC 41 ⟩;
⟨ Set inc_src and dec_src bits; ELTE 26 ⟩;
for (k = 0; k < 1; k++)      /* ELTE */
  do4(source[k], OR,
        make2(AND, imm, mem[k < 4 ? k : 3]),
        make3(AND, rel, comp(trap), inc_dest[k]),
        make2(AND, dirind, old_src[k]),
        make2(AND, extra, mem[k]));
for ( ; k < 4; k++)      /* ELTE */
  do5(source[k], OR,
        make2(AND, imm, mem[k < 4 ? k : 3]),
        make2(AND, trap, mem[k − 1]),
        make3(AND, rel, comp(trap), inc_dest[k]),
        make2(AND, dirind, old_src[k]),
        make2(AND, extra, mem[k]));
for ( ; k < 5; k++)      /* ELTE */
  do5(source[k], OR,
        make3(AND, imm, comp(load_imm), mem[k < 4 ? k : 3]),
        make2(AND, trap, mem[k − 1]),
        make3(AND, rel, comp(trap), inc_dest[k]),
        make2(AND, dirind, old_src[k]), make2(AND, extra, mem[k]));
for ( ; k < 6; k++)      /* ELTE */
  do5(source[k], OR, trap,
        make3(AND, imm, comp(load_imm), mem[k < 4 ? k : 3]),
        make3(AND, rel, comp(trap), inc_dest[k]),
        make2(AND, dirind, old_src[k]),
        make2(AND, extra, mem[k]));
for ( ; k < 16; k++)      /* ELTE */
  do4(source[k], OR,
        make3(AND, imm, comp(load_imm), mem[k < 4 ? k : 3]),
        make3(AND, rel, comp(trap), inc_dest[k]),
        make2(AND, dirind, old_src[k]),
        make2(AND, extra, mem[k]));
```

This code is used in section 17.

23. ⟨ Local variables for *risc_elte* 9 ⟩ +≡

```
Vertex *up, *down;   /* gates used when computing inc_src, etc.; ELTE moved */
Vertex *dest_match[16];   /* dest_match[r] ≡ 1 iff DST = r */
Vertex *old_dest[16];   /* contents of destination register before operation */
Vertex *src_match[16];   /* src_match[r] ≡ 1 iff SRC = r; ELTE */
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 *matrix[16];   /* result of matrix operation; ELTE */
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 *inc_src[16];   /* contents of register SRC, plus 1; ELTE */
Vertex *dec_src[16];   /* contents of register SRC, minus 1; ELTE */
Vertex *result[18];   /* result of operating on old_dest and source */
```

24. 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 24 ⟩ ≡

```
for (r = 0; r < regs; r++) {
    t6 = make2(write, comp(cond));
   /* ELTE */
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);
  do2(t5, OR,
          /* ELTE */
  make2(AND, comp(brk), old_dest[k]),
          /* ELTE */
  make2(AND, brk, reg[0] + k));   /* ELTE */
  { register Arc *a = gb_virgin_arc();   /* ELTE */
    a→tip = t5;   /* ELTE */
    a→next = new_graph→outs;   /* ELTE */
    new_graph→outs = a;   /* pointer to memory address bit; ELTE */
  }
}   /* arcs for output write data bits will appear in big-endian order; ELTE */
```

This code is used in section 22.

25. ⟨ Set *old_src* to the present value of the source register 25 ⟩ ≡

```
for (r = 0; r < regs; r++)    /* ELTE */
  do4(src_match[r], AND, even_comp(r, src[0]), even_comp(r >> 1, src[1]),
      even_comp(r >> 2, src[2]), even_comp(r >> 3, src[3]));    /* ELTE */
for (k = 0; k < 16; k++) {
  for (r = 0; r < regs; r++)
    tmp[r] = make2(AND, src_match[r], reg[r] + k);    /* ELTE */
  old_src[k] = new_vert(OR);    /* ELTE */
  for (r = 0; r < regs; r++) gb_new_arc(old_src[k], tmp[r], DELAY);
}
```

This code is used in section 22.

26. ⟨ Set *inc_src* and *dec_src* bits; ELTE 26 ⟩ ≡

```
inc_src[0] = comp(old_src[0]); dec_src[0] = comp(old_src[0]);
up = old_src[0];    /* remaining bits must increase */
down = comp(old_src[0]);    /* remaining bits must decrease */
for (k = 1; ; k++) {
  comp(up); comp(down);
  do2(inc_src[k], OR,
      make2(AND, comp(old_src[k]), up),
      make2(AND, old_src[k], comp(up)));
  do2(dec_src[k], OR,
      make2(AND, comp(old_src[k]), down),
      make2(AND, old_src[k], comp(down)));
  if (k < 15) {
    up = make2(AND, up, old_src[k]);
    down = make2(AND, down, comp(old_src[k]));
  } else break;
}
```

This code is used in section 22.

27. ⟨ Create gates for output write address and flags; `ELTE` 27 ⟩ ≡

```
start_prefix("W");
for (k = 0; k < 4; k++) {
  do4(t5, OR, make3(AND, imm, comp(brk), mem[k]),
      make3(AND, rel, comp(brk), dec_src[k]),
      make3(AND, dir, comp(brk), old_src[k]),
      make3(AND, extra, comp(brk), mem[k]));
  { register Arc *a = gb_virgin_arc();
    a→tip = t5;
    a→next = new_graph→outs;
    new_graph→outs = a;    /* pointer to write address bit */
  }
}    /* arcs for write address bits will appear in big-endian order */
for ( ; k < 16; k++) {
  do3(t5, OR, make3(AND, rel, comp(brk), dec_src[k]),
      make3(AND, dir, comp(brk), old_src[k]),
      make3(AND, extra, comp(brk), mem[k]));
  { register Arc *a = gb_virgin_arc();
    a→tip = t5;
    a→next = new_graph→outs;
    new_graph→outs = a;    /* pointer to write address bit */
  }
}    /* arcs for write address bits will appear in big-endian order */
{ register Arc *a = gb_virgin_arc();
  a→tip = make3(OR, brk, comp(rel), nzs);
  a→next = new_graph→outs;
  new_graph→outs = a;    /* pointer to write data flag */
}
{ register Arc *a = gb_virgin_arc();
  a→tip = extra;
  a→next = new_graph→outs;
  new_graph→outs = a;    /* pointer to extra flag */
}
do5(t5, OR, brk, make3(AND, write, comp(cond), imm),
    make3(AND, write, comp(cond), rel),
    make3(AND, write, comp(cond), dir),
    make3(AND, write, comp(cond), extra));
{ register Arc *a = gb_virgin_arc();
  a→tip = t5;
  a→next = new_graph→outs;
  new_graph→outs = a;    /* pointer to write flag */
}
```

This code is used in section 17.

28. ⟨ Create gates for the general logic operation 28 ⟩ ≡

```
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.

29. ⟨ Create gates for the conditional load operations 29 ⟩ ≡

```
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.

30. ⟨ Local variables for *risc_elte* 9 ⟩ +≡

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

31. 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 31 ⟩ ≡

```
start_prefix("Z");
⟨ Create gates for the next_loc and next_next_loc bits 32 ⟩;
⟨ Create gates for the result bits 33 ⟩;
⟨ Create gates for the new values of registers 1 to regs 35 ⟩;
⟨ Create gates for the new values of flags; ELTE 36 ⟩;
⟨ Create gates for the new values of the program register 34 ⟩;
⟨ Create gates for the new values of register 0 and the memory address register 38 ⟩;
```

This code is used in section 17.

32. ⟨ Create gates for the *next_loc* and *next_next_loc* bits 32 ⟩ ≡

```
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 31.

33. ⟨ Create gates for the *result* bits 33 ⟩ ≡

```
for (k = 0; k < 5; k++) {
   do5(result[k], OR,
         make2(AND, mixed, dir, tmp[k + 2]),
         make5(AND, mod[3], mod[2], comp(mod[1]), comp(mod[0]), matrix[k]),
         make2(AND, comp(mod[3]), shift[k]),
         make4(AND, mod[3], comp(mod[2]), comp(mod[1]), sum[k]),
         make4(AND, mod[3], comp(mod[2]), mod[1], diff[k]));
   do5(result[k], OR,
         make2(AND, comp(op), log[k]),
         make2(AND, jump, next_loc[k]),
         make2(AND, trap, next_loc[k]),
         make3(AND, cond, change, source[k]),
         make3(AND, op, comp(cond), result[k]));
}
for ( ; k < 16; k++) {
   do4(result[k], OR,
         make5(AND, mod[3], mod[2], comp(mod[1]), comp(mod[0]), matrix[k]),
         make2(AND, comp(mod[3]), shift[k]),
         make4(AND, mod[3], comp(mod[2]), comp(mod[1]), sum[k]),
         make4(AND, mod[3], comp(mod[2]), mod[1], diff[k]));
   do5(result[k], OR,
         make2(AND, comp(op), log[k]),
         make2(AND, jump, next_loc[k]),
         make2(AND, trap, next_loc[k]),
         make3(AND, cond, change, source[k]),
         make3(AND, op, 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 31.

34. 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 34 ⟩ ≡

```
for (k = 0; k < 10; k++) latchit(mem[k + 6], prog + k);
```

This code is used in section 31.

35. ⟨ Create gates for the new values of registers 1 to *regs* 35 ⟩ ≡

```
t8 = make2(AND, change, comp(ind));      /* should destination register change? */
for (r = 1; r < regs; r++) {
   t7 = make2(AND, t8, dest_match[r]);      /* should destination register r change? */
   do2(t6, OR,
            /* should source register r change? ELTE */
   make2(AND, push, src_match[r]),
       make2(AND, pop, src_match[r]));
   for (k = 0; k < 16; k++) {
      do4(t5, OR,
          make2(AND, t7, result[k]),
          make4(AND, comp(t7), t6, push, dec_src[k]),
          make4(AND, comp(t7), t6, pop, inc_src[k]),
          make3(AND, comp(t7), comp(t6), reg[r] + k));
      latchit(t5, reg[r] + k);
   }
}
```

This code is used in section 31.

36. ⟨ Create gates for the new values of flags; ELTE 36 ⟩ ≡

```
do2(t5, OR,
      make2(AND, sign, nochangeflags),
      make2(AND, result[15], comp(nochangeflags)));
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])));
do2(t5, OR,
      make2(AND, nonzero, nochangeflags),
      make2(AND, t5, comp(nochangeflags)));
latchit(t5, nonzero);
do3(t5, OR,
      make2(AND, overflow, nochangeflags),
      make2(AND, overflow, comp(op)),
      make3(AND, result[17], comp(nochangeflags), op));
latchit(t5, overflow);
do3(t5, OR,
      make2(AND, carry, nochangeflags),
      make2(AND, carry, comp(op)),
      make3(AND, result[16], comp(nochangeflags), op));
latchit(t5, carry);
do2(nextra, OR, make2(AND, ind, comp(cond)), make2(AND, ind, change));
latchit(nextra, extra);
do4(t5, OR, xresume, resume,
      make3(AND, enabled, comp(brk), comp(trap)),
      make5(AND, nope, comp(cond), rel, nzd, comp(nzs)));
latchit(t5, enabled); do3 (ntranslate, OR, xresume, resume, make3(AND, translate, comp(brk), comp(trap));
      latchit(ntranslate, translate); latchit(resume0, xresume);
```

This code is used in section 31.

37. ⟨Local variables for *risc_elte* 9⟩ +≡
  Vertex ∗*nextra*; /∗ must we take an extra cycle? ∗/
  Vertex ∗*ntranslate*; /∗ must we translate in the next cycle? ∗/

38. 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 38 ⟩ ≡
  skip = make2(AND, cond, comp(change));      /* false conditional? */
  do2(hop, OR, trap, make2(AND, comp(cond), jump));      /* JUMP or TRAP 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));
  do3(special, OR, load_imm, resume0,
              /* read from data memory; ELTE */
  make3(AND, comp(skip), ind, nzs));      /* address in source; ELTE */
  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));      /* ELTE */
    latchit(t4, reg[0] + k);
    do2(t4, OR,
              /* ELTE */
    make2(AND, special, comp(resume0), source[k]),
          make2(AND, comp(special), t5));
    { register Arc *a = gb_virgin_arc();
      if (k)      /* ELTE */
        a→tip = make3(AND, t4, run_bit, comp(brk));
      else {
        do2(t5, OR, make2(AND, run_bit, brk),
              make3(AND, t4, run_bit, comp(brk)));
        a→tip = t5;
      }
      a→next = new_graph→outs;
      new_graph→outs = a;      /* pointer to memory address bit */
    }
  }      /* arcs for output bits will appear in big-endian order */
  { register Arc *a = gb_virgin_arc();      /* ELTE */
    a→tip = make3(AND, special, run_bit, comp(brk));
    a→next = new_graph→outs;
    new_graph→outs = a;      /* pointer to special flag: data address space */
  }
  { register Arc *a = gb_virgin_arc();      /* ELTE */
    a→tip = make2(AND, ntranslate, run_bit);
    a→next = new_graph→outs;
    new_graph→outs = a;      /* pointer to ntranslate flag */
  }
  { register Arc *a = gb_virgin_arc( );      /* ELTE */
    a→tip = make2(AND, nextra, run_bit);
    a→next = new_graph→outs;
    new_graph→outs = a;      /* pointer to nextra value */
  }
```

This code is used in section 31.

39. ⟨ Local variables for *risc_elte* 9 ⟩ +≡

```
Vertex *skip;      /* are we skipping a conditional load operation? */
Vertex *hop;      /* are we doing a JUMP or a TRAP? */
Vertex *normal;      /* is this a case where register 0 is simply incremented? */
Vertex *special;      /* is this a case where the memory address register point to data space? ELTE */
```

40. Serial addition. We haven't yet specified the parts of *risc_elte* 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;
  }
```

41. 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 41 ⟩ ≡
  make_adder(4L, 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.

42. 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 42 ⟩ ≡
  start_prefix("A");
  ⟨ Create gates for the shift operations 45 ⟩;
  make_adder(16L, old_dest, source, sum, make2(AND, carry, mod[0]), 1);      /* adder */
  make_adder(16L, 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.

43.
```
⟨ Create gates for the matrix operation; ELTE 43 ⟩ ≡
  for (i = 0; i < 4; i++)
    for (j = 0; j < 4; j++) {
      t1 = make2(AND, old_dest[0 + i], source(4 * j + 0));
      t2 = make2(AND, old_dest[4 + i], source(4 * j + 1));
      t3 = make2(AND, old_dest[8 + i], source(4 * j + 2));
      t4 = make2(AND, old_dest[12 + i], source(4 * j + 3));
      t5 = make_xor(t1, t2);
      t6 = make_xor(t3, t4);
      matrix[4 * j + i] = make_xor(t5, t6);
    }
```

This code is used in section 17.

44. Sideway add operation use a three-adder; `ELTE`. First we define this. (There is a much better construction for sideway add, find by Ádám Nagy; this simple version will be substituted by his version in the future.)

```
#define three_add(result_high, result_low, v1, v2, v3) { t6 = v1; t7 = v2; t8 = v3; comp(t6);
            comp(t7); comp(t8);      /* generate inverse gates */
            do4(result_low, OR,
                  make3(AND, t6, comp(t7), comp(t8)),
                  make3(AND, comp(t6), t7, comp(t8)),
                  make3(AND, comp(t6), comp(t7), t8),
                  make3(AND, t6, t7, t8)); do3 (result_high, OR,
                  make2(AND, t6, t7),
                  make2(AND, t7, t8),
                  make2(AND, t8, t6); }
```

⟨ Create gates for the sideway add operation; `ELTE` 44 ⟩ ≡

```
   three_add(tmp[15], tmp[2], old_src[0], old_src[1], old_src[2]);
   three_add(tmp[14], tmp[3], old_src[3], old_src[4], old_src[5]);
   three_add(tmp[13], tmp[4], old_src[6], old_src[7], old_src[8]);
   three_add(tmp[12], tmp[5], old_src[9], old_src[10], old_src[11]);
   three_add(tmp[11], tmp[6], old_src[12], old_src[13], old_src[14]);
   three_add(tmp[10], tmp[7], old_src[15], tmp[2], tmp[3]);
   three_add(tmp[9], tmp[3], tmp[4], tmp[5], tmp[6]);
   tmp[2] = make_xor(tmp[7], tmp[3]);
   tmp[8] = make2(AND, tmp[3], tmp[7]);
   three_add(tmp[6], tmp[7], tmp[15], tmp[14], tmp[13]);
   three_add(tmp[5], tmp[15], tmp[12], tmp[11], tmp[10]);
   three_add(tmp[4], tmp[14], tmp[9], tmp[8], tmp[7]);
   tmp[3] = make_xor(tmp[15], tmp[14]);
   tmp[7] = make2(AND, tmp[15], tmp[14]);
   three_add(tmp[15], tmp[8], tmp[4], tmp[5], tmp[6]);
   tmp[4] = make_xor(tmp[7], tmp[8]);
   tmp[14] = make2(AND, tmp[7], tmp[8]);
   tmp[5] = make_xor(tmp[15], tmp[14]);
   tmp[6] = make2(AND, tmp[15], tmp[14]);
```

This code is used in section 17.

45. ⟨ Create gates for the shift operations 45 ⟩ ≡

```
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 42.

46. 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_elte routine 46⟩ ≡
  long run_risc_elte(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 47⟩;
    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 49⟩;
      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 48⟩;
    ⟨Dump the register contents into risc_state 50⟩;
    return 0;
  }
```

This code is used in section 7.

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

```
⟨Print a headline 47⟩ ≡
  {
    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 46.

48. ⟨ Print a footline 48 ⟩ ≡

```
printf("Execution␣terminated␣with␣memory␣address␣%04lx.\n", l);
```

This code is used in section 46.

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

⟨ Print register contents 49 ⟩ ≡

```
{
  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 46.

50. ⟨ Dump the register contents into *risc_state* 50 ⟩ ≡

```
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 46.

51. ⟨ Global variables 51 ⟩ ≡

```
unsigned long risc_state[18];
```

This code is used in section 7.

52. 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 52⟩ ≡
  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.

53. ⟨gb_gates_elte.h 1⟩ +≡

```
#define bit z.I
```

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

a: 3, 24, 27, 38, 46, 52.
add: 40.
alloc_fault: 8.
alt: 2, 3, 34, 49, 50, 52.
`AND`: 2, 6, 15, 21, 22, 24, 25, 26, 27, 28, 29, 32, 33, 34, 35, 36, 38, 40, 41, 42, 43, 44, 45.
Arc: 2, 3, 24, 27, 38, 46, 52.
arcs: 6, 52.
bar: 14.
bit: 52, 53.
brk: 20, 21, 24, 27, 36, 38.
c: 46.
carry: 18, 19, 29, 36, 40, 42, 49, 50.
change: 29, 30, 33, 35, 36, 38.
comp: 14, 15, 21, 22, 24, 26, 27, 28, 29, 32, 33, 35, 36, 38, 40, 41, 42, 44, 45.
cond: 20, 21, 24, 27, 29, 33, 36, 38.
count: 11, 12, 19.
dec_src: 23, 26, 27, 35.
`DELAY`: 13, 24, 25.
dest: 20, 21, 24.
dest_match: 23, 24, 35.
diff: 23, 33, 42.
dir: 20, 21, 27, 33.
dirind: 20, 21, 22.
down: 23, 26, 41.
do2: 20, 21, 24, 26, 35, 36, 38, 42.
do3: 20, 21, 27, 29, 33, 36, 38, 40, 41, 44, 45.
do4: 20, 22, 24, 25, 27, 28, 29, 33, 35, 36, 38, 40, 44, 45.
do5: 20, 22, 27, 33.
enable: 19.
enabled: 18, 21, 36.
even_comp: 14, 24, 25, 40.
extra: 18, 19, 21, 22, 27, 34, 36, 47, 49, 50.
first_of: 19.
g: 3, 46, 52.
gate_eval: 1, 3, 10, 46.
gb_new_arc: 13, 14, 24, 25.
gb_new_graph: 16.
gb_recycle: 8.
gb_save_string: 11, 14.
gb_trouble_code: 8.
gb_virgin_arc: 24, 27, 38.
Graph: 1, 3, 8, 9, 46, 52.
hop: 38, 39.
i: 9.
id: 16.
imm: 20, 21, 22, 27.
impossible: 17.
in_vec: 3, 4.
inc_dest: 22, 23, 41.
inc_src: 23, 26, 35.
ind: 20, 21, 34, 35, 36, 38.
int_bit: 18, 21.
interupt_bit: 19.
is_boolean: 2, 52.
j: 9.
jump: 20, 21, 33, 38.
k: 9, 40, 46.
l: 46.
latch: 34.
latchit: 34, 35, 36, 38.
load_imm: 20, 21, 22, 38.
log: 23, 28, 33.
m: 46.
make_adder: 40, 41, 42.
make_xor: 15, 32, 40, 43, 44, 45.
make2: 13, 15, 20, 21, 22, 24, 25, 26, 29, 32, 33, 34, 35, 36, 38, 40, 41, 42, 43, 44, 45.
make3: 13, 20, 21, 22, 27, 28, 29, 33, 35, 36, 38, 40, 41, 42, 44, 45.
make4: 13, 20, 21, 33, 35, 36, 38, 45.
make5: 13, 20, 21, 33, 36, 38, 45.
matrix: 23, 33, 43.
mem: 18, 19, 21, 22, 27, 34, 41.
mixed: 20, 21, 33.
mod: 20, 21, 28, 29, 33, 36, 42, 45.
n: 40, 46.
name: 11, 14, 52.
name_buf: 11, 12, 14.
new_graph: 8, 9, 16, 17, 24, 27, 38.
new_vert: 11, 13, 19, 24, 25.
next: 5, 6, 24, 27, 38, 46, 52.
next_loc: 23, 32, 33, 38.
next_next_loc: 23, 32, 34, 38.
next_vert: 11, 12, 14, 16, 17.
nextra: 36, 37, 38.
no_room: 16.
nochangeflags: 20, 21, 36.
nonzero: 18, 19, 29, 36, 49, 50.
nope: 20, 21, 36.
normal: 38, 39.
`NOT`: 2, 6, 14.
ntranslate: 36, 37, 38.
numeric_prefix: 12, 19.
nzd: 21, 36, 38.
nzs: 20, 21, 27, 36, 38.
o: 46.
old_dest: 23, 24, 28, 41, 42, 43.
old_src: 22, 23, 24, 25, 26, 27, 44.
op: 20, 21, 29, 33, 36.

OR: 2, 6, 15, 21, 22, 24, 25, 26, 27, 28, 29, 33, 35, 36, 38, 40, 41, 42, 44, 45.
out_vec: 3, 5.
outs: 2, 5, 24, 27, 38, 46, 52.
overflow: 18, 19, 29, 36, 49, 50.
p_gates: 1, 52.
page_bit: 18, 19, 21.
panic: 8, 16, 17.
panic_code: 8.
pop: 20, 21, 35.
pr_gate: 52.
prefix: 11, 12, 19.
print_gates: 1, 52.
printf: 47, 48, 49, 52.
prod: 2, 40.
prog: 18, 19, 21, 34, 47, 49, 50.
push: 20, 21, 35.
r: 9, 46.
reg: 18, 19, 24, 25, 32, 35, 38.
regs: 8, 16, 19, 24, 25, 35.
rel: 20, 21, 22, 27, 36.
result: 20, 23, 33, 35, 36, 38.
result_high: 44.
result_low: 44.
resume: 20, 21, 36.
resume0: 20, 21, 36, 38.
risc: 2, 9, 46.
risc_elte: 1, 8, 40.
risc_state: 1, 46, 50, 51.
rom: 46, 49.
run_bit: 18, 19, 34, 38.
run_risc: 46.
run_risc_elte: 1, 46.
s: 46.
shift: 23, 33, 45.
sign: 18, 19, 29, 36, 49, 50.
size: 46, 49.
skip: 38, 39.
source: 22, 23, 28, 33, 38, 42, 43, 45.
special: 38, 39.
sprintf: 11, 12, 14, 16.
src: 25.
src_match: 23, 25, 35.
start_prefix: 12, 19, 21, 22, 27, 28, 29, 31, 42.
strcpy: 12, 16, 19.
sum: 23, 33, 42.
t: 3, 11, 13.
the_boolean: 2, 52.
three_add: 44.
tip: 2, 5, 6, 24, 27, 38, 46, 52.
tip_value: 2, 5.
tmp: 20, 24, 25, 29, 33, 44.
trace_regs: 46, 47, 49.
translate: 18, 19, 36.
trap: 20, 21, 22, 33, 36, 38.
typ: 2, 3, 11, 14, 49, 50, 52.
t1: 15, 20, 40, 43.
t2: 15, 20, 40, 43.
t3: 20, 40, 43.
t4: 20, 38, 40, 43.
t5: 20, 24, 27, 32, 35, 36, 38, 43.
t6: 20, 24, 35, 43, 44.
t7: 20, 35, 44.
t8: 20, 35, 44.
u: 14, 15.
up: 23, 26, 41.
util_types: 16.
v: 3, 11, 13, 14, 15, 46, 52.
val: 2, 3, 4, 6, 46, 49, 50.
Vertex: 3, 11, 12, 13, 14, 15, 18, 20, 23, 30, 37, 39, 40, 46, 52.
vertices: 3, 4, 16, 17, 46, 49, 50, 52.
v1: 13, 20, 44.
v2: 13, 20, 44.
v3: 13, 20, 44.
v4: 13, 20.
v5: 13, 20.
write: 20, 21, 24, 27.
x: 40, 46.
XOR: 2, 6.
xresume: 18, 19, 36.
y: 40.
z: 40.
zz: 2.

⟨ Add the RISC data to *new_graph* 17 ⟩ Used in section 8.
⟨ Compute the value t of a classical logic gate 6 ⟩ Used in section 3.
⟨ 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 output write address and flags; `ELTE` 27 ⟩ Used in section 17.
⟨ Create gates for the arithmetic operations 42 ⟩ Used in section 17.
⟨ Create gates for the conditional load operations 29 ⟩ Used in section 17.
⟨ Create gates for the general logic operation 28 ⟩ Used in section 17.
⟨ Create gates for the matrix operation; `ELTE` 43 ⟩ Used in section 17.
⟨ Create gates for the new values of flags; `ELTE` 36 ⟩ Used in section 31.
⟨ Create gates for the new values of register 0 and the memory address register 38 ⟩ Used in section 31.
⟨ Create gates for the new values of registers 1 to *regs* 35 ⟩ Used in section 31.
⟨ Create gates for the new values of the program register 34 ⟩ Used in section 31.
⟨ Create gates for the shift operations 45 ⟩ Used in section 42.
⟨ Create gates for the sideway add operation; `ELTE` 44 ⟩ Used in section 17.
⟨ Create gates for the *next_loc* and *next_next_loc* bits 32 ⟩ Used in section 31.
⟨ Create gates for the *result* bits 33 ⟩ Used in section 31.
⟨ Create gates that bring everything together properly 31 ⟩ Used in section 17.
⟨ Create the inputs and latches 19 ⟩ Used in section 17.
⟨ Dump the register contents into *risc_state* 50 ⟩ Used in section 46.
⟨ Global variables 51 ⟩ 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, 40 ⟩ Used in section 7.
⟨ Local variables for *risc_elte* 9, 18, 20, 23, 30, 37, 39 ⟩ Used in section 8.
⟨ Print a footline 48 ⟩ Used in section 46.
⟨ Print a headline 47 ⟩ Used in section 46.
⟨ Print register contents 49 ⟩ Used in section 46.
⟨ Private variables 12 ⟩ Used in section 7.
⟨ Read a sequence of input values from *in_vec* 4 ⟩ Used in section 3.
⟨ Set *inc_dest* to *old_dest* plus `SRC` 41 ⟩ Used in section 22.
⟨ Set *inc_src* and *dec_src* bits; `ELTE` 26 ⟩ Used in section 22.
⟨ Set *old_dest* to the present value of the destination register 24 ⟩ Used in section 22.
⟨ Set *old_src* to the present value of the source register 25 ⟩ 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 *print_gates* routine 52 ⟩ Used in section 7.
⟨ The *risc_elte* routine 8 ⟩ Used in section 7.
⟨ The *run_risc_elte* routine 46 ⟩ Used in section 7.
⟨ `gb_gates_elte.h` 1, 2, 53 ⟩

November 17, 2013 at 11:27

GB_GATES_ELTE

	Section	Page
Introduction	1	1
The RISC ELTE netlist	8	5
Serial addition	40	26
RISC management	46	30
Generalized gate graphs	52	32
Index	54	33

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