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

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

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

To run the program under UNIX, say '`test_risc_elte` ⟨trace⟩', where ⟨trace⟩ is nonempty if and only if you want the machine computations to be printed out.

The program will prompt you for two numbers, and it will use the simulated RISC ELTE machine to compute their product, quotient and some other things. Then it will ask for two more numbers, and so on.

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

```
#include "gb_graph.h"      /* the standard GraphBase data structures */
#include "gb_gates_elte.h"      /* routines for gate graphs */
  ⟨ Preprocessor definitions ⟩
  ⟨ Global variables 3 ⟩
  main(argc, argv)
      int argc;      /* the number of command-line arguments */
      char *argv[];      /* an array of strings containing those arguments */
{
    trace = (argc > 1 ? 8 : 0);      /* we'll show registers 0–7 if tracing */
    if ((g = risc_elte(8L)) ≡ Λ) {
      printf("Sorry, I couldn't generate the graph (trouble code %ld)!\n", panic_code);
      return (−1);
    }
    printf("Welcome to the world of microRISC ELTE.\n");
    while (1) {
      ⟨ Prompt for two numbers; break if unsuccessful 4 ⟩;
      ⟨ Test some instructions of the RISC ELTE machine 9 ⟩;
      ⟨ Use the RISC ELTE machine to compute the product, p 7 ⟩;
      printf("The product of %ld and %ld is %ld%s.\n", m, n, p, o ? " (overflow occurred)" : "");
      ⟨ Use the RISC ELTE machine to compute the quotient and remainder, q and r 8 ⟩;
      printf("The quotient is %ld, and the remainder is %ld.\n", q, r);
    }
    return 0;      /* normal exit */
}
```

3. ⟨ Global variables 3 ⟩ ≡

```
Graph *g;      /* graph that defines a simple RISC machine */
long o, p, q, r;      /* overflow, product, quotient, remainder */
long trace;      /* number of registers to trace */
long m, n;      /* numbers to be multiplied and divided */
char buffer[100];      /* input buffer */
```

See also section 6.

This code is used in section 2.

4. **#define** *prompt*(*s*)
{ *printf*(*s*); *fflush*(*stdout*); /* make sure the user sees the prompt */
if (*fgets*(*buffer*, 99, *stdin*) ≡ Λ) **break**; }

⟨ Prompt for two numbers; **break** if unsuccessful 4 ⟩ ≡
prompt("\nGimme␣a␣number:␣");
step0:
if (*sscanf*(*buffer*, "%ld", &*m*) ≠ 1) **break**;
step1:
if (*m* ≤ 0) {
prompt("Excuse␣me,␣I␣meant␣a␣positive␣number:␣");
if (*sscanf*(*buffer*, "%ld", &*m*) ≠ 1) **break**;
if (*m* ≤ 0) **break**;
}
while (*m* > #7fff) {
prompt("That␣number's␣too␣big;␣please␣try␣again:␣");
if (*sscanf*(*buffer*, "%ld", &*m*) ≠ 1) **goto** *step0*; /* *step0* will **break** out */
if (*m* ≤ 0) **goto** *step1*;
}
⟨ Now do the same thing for *n* instead of *m* 5 ⟩;

This code is used in section 2.

5. ⟨ Now do the same thing for *n* instead of *m* 5 ⟩ ≡
prompt("OK,␣now␣gimme␣another:␣");
if (*sscanf*(*buffer*, "%ld", &*n*) ≠ 1) **break**;
step2:
if (*n* ≤ 0) {
prompt("Excuse␣me,␣I␣meant␣a␣positive␣number:␣");
if (*sscanf*(*buffer*, "%ld", &*n*) ≠ 1) **break**;
if (*n* ≤ 0) **break**;
}
while (*n* > #7fff) {
prompt("That␣number's␣too␣big;␣please␣try␣again:␣");
if (*sscanf*(*buffer*, "%ld", &*n*) ≠ 1) **goto** *step0*; /* *step0* will **break** out */
if (*n* ≤ 0) **goto** *step2*;
}

This code is used in section 4.

6. A RISC program. Here is the little program we will run on the little computer. It consists mainly of a subroutine called *tri*, which computes the value of the ternary operation $x\lfloor y/z\rfloor$, assuming that $y \geq 0$ and $z > 0$; the inputs x, y, z appear in registers $1, 2, 3$, respectively, and the exit address is assumed to be in register 7. As special cases we can compute the product xy (letting $z = 1$) or the quotient $\lfloor y/z\rfloor$ (letting $x = 1$). When the subroutine returns, it leaves the result in register 4; it also leaves the value $(y \bmod z) - z$ in register 2. Overflow will be set if and only if the true result was not between -2^{15} and $2^{15} - 1$, inclusive.

It would not be difficult to modify the code to make it work with unsigned 16-bit numbers, or to make it deliver results with 32 or 48 or perhaps even 64 bits of precision.

```
#define div 112 /* location 'div' in the program below */
#define mult 115 /* location 'mult' in the program below */
#define test 16 /* location 'test' in the program below */
#define memry_size 144 /* the number of instructions in the program below */
```

⟨ Global variables 3 ⟩ +≡

```
unsigned long memry[memry_size] = {  /* a "ram memory" used by run_risc */
  #0f87,  /* 0: cold start, jump to 7 */
  #0f70,  /*   jumpto interrupt handler */
  #0058,  /*   at 88 */
  #0000,  /* empty */
  #0000,  /* empty */
  #0000,  /* empty */
  #0000,  /* empty */
  #1ff0,  /* start: r1 = m (contents of next word) */
  #1111,  /* (we will put the value of m here, in memry[1]) */
  #2a30,  /*   r2 = n (contents of next word) */
  #3333,  /* (we will put the value of n here, in memry[3]) */
  #7f70,  /*   jumpto (contents of next word), r7 = return address */
  #5555,  /* (we will put an address here, in memry[5]) */
  #0f8f,  /* halt without changing any status bits */
  #0000,  /* empty */
  #0000,  /* empty */
  #2d50,  /* 16: push n to stack pointer 0 */
  #2d70,  /*   write n to given address */
  #000e,  /*   14 */
  #1d46,  /*   write m to address 6 */
  #2d51,  /*   push n to stack pointer m */
  #1d62,  /*   write m to address n */
  #1d72,  /*   write m indirect address at n */
  #3e71,  /*   pop n from stack pointed by m into register 3 */
  #3c61,  /* 24: mxor of n with m */
  #4e46,  /*   load from address 6 to register 4 */
  #4e61,  /*   sideway add register 1 to 4 */
  #4e70,  /*   pop from next address into register 4 */
  #1111,  /*   data */
  #4ad0,  /*   NOPE */
  #5e5f,  /*   TRAP 15, return address into register 5 */
  #0fa7,  /*   return (literally, r0 = r7) */
  #0000,  /* 32: empty */
  #0000,  /* empty */
  #0000,  /* empty */
  #0000,  /* empty */
  #0000,  /* empty */
  #0000,  /* empty */
```

```
#0000,    /* empty */
#0000,    /* empty */
#0000,    /* 40: empty */
#0000,    /* empty */
#0000,    /* empty */
#0000,    /* empty */
#0000,    /* empty */
#0000,    /* empty */
#0000,    /* empty */
#0000,    /* empty */
#0000,    /* 48: empty */
#0000,    /* empty */
#0000,    /* empty */
#0000,    /* empty */
#0000,    /* empty */
#0000,    /* empty */
#0000,    /* empty */
#0000,    /* empty */
#0000,    /* 56: empty */
#0000,    /* empty */
#0000,    /* empty */
#0000,    /* empty */
#0000,    /* empty */
#0000,    /* empty */
#0f70,    /*   jump without link */
#0050,    /*   to 80 */
#624e,    /* 64: as well as this curious constant setting by shift */
#3c51,    /*   mor of n with m */
#4ad0,    /*   NOPE */
#0f70,    /*   jump back */
#0052,    /*   to 82 */
#0000,    /* empty */
#0000,    /* empty */
#0000,    /* empty */
#0000,    /* 72: empty */
#0000,    /* empty */
#0000,    /* empty */
#0000,    /* empty */
#0000,    /* empty */
#0000,    /* empty */
#0000,    /* empty */
#0000,    /* empty */
#0244,    /* 80: fast jump done, but this may affect flags */
#0000,    /* empty */
#6aa0,    /*   get present address into register 6 */
#6a94,    /*   and add 4 to write into memory at address 0 */
#6d40,    /*   to prepare simulation of interrupt after */
#0f81,    /*   this instruction, which jump to address 1 */
#0e55,    /*   resume by link in register 5 */
#0000,    /* empty */
#1d43,    /* 88: int handler: save register 1 at address 3 */
#1f80,    /*   clear register 1 */
```

```
#1c88,    /*   save S flag into register 1 */
#1a94,    /*   save N flag into register 1 */
#1cd2,    /*   save K flag into register 1 */
#1ad1,    /*   save V flag into register 1 */
#2d44,    /*   save register 2 at address 4; etc. */
#2e44,    /*   load register 2 from address 4 */
#1061,    /* 96: shift register 1 to left by 1 */
#1061,    /*   shift register 1 to left by 1 */
#1621,    /*   shift register 1 to right by 4 to set K and V */
#1830,    /*   make and of register 1 by */
#8001,    /*   this mask to set S and N */
#0f8f,    /*   halt without changing any status bits */
#0e50,    /*   resume 0 */
#0000,    /* empty */
#0000,    /* 104: empty */
#0000,    /* empty */
#0000,    /* empty */
#0000,    /* empty */
#0000,    /* empty */
#0000,    /* empty */
#0000,    /* empty */
#0000,    /* empty */
#3a21,    /* div: r3 = r1 */
#1a01,    /*   r1 = 1 */
#0a13,    /*   goto tri (literally, r0 += 4) */
#3a01,    /* mult: r3 = 1 */
#0a11,    /*   goto tri (literally, r0 += 1) */
#4000,    /* tri: r4 = 0 */
#5000,    /*   r5 = 0 */
#6000,    /*   r6 = 0 */
#2a63,    /*   r2 -= r3 */
#0f95,    /*   goto l2 */
#3063,    /* l1: r3 <<= 1 */
#1061,    /*   r1 <<= 1 */
#6ac1,    /*   if (overflow) r6 = 1 */
#5fd1,    /*   r5++ */
#2a63,    /* l2: r2 -= r3 */
#039b,    /*   if (≥ 0) goto l1 */
#0843,    /*   goto l4 */
#3463,    /* l3: r3 >>= 1 */
#1561,    /*   r1 >>= 1 */
#2863,    /* l4: r2 += r3 */
#0c94,    /*   if (< 0) goto l5 */
#4861,    /*   r4 += r1 */
#6ac1,    /*   if (overflow) r6 = 1 */
#2a63,    /*   r2 -= r3 */
#5a41,    /* l5: r5-- */
#0398,    /*   if (≥ 0) goto l3 */
#6666,    /*   if (r6) force overflow (literally r6 >>= 4) */
#0fa7,    /*   return (literally, r0 = r7, preserving overflow) */
#0000,    /* empty */
#0000,    /* empty */
```

```
#0000,      /* empty */
#0000};     /* empty */
```

7. ⟨ Use the RISC ELTE machine to compute the product, p 7 ⟩ ≡

```
memry[0] = #f87;
memry[8] = m;
memry[10] = n;
memry[12] = mult;
run_risc_elte(g, memry, memry_size, trace);
p = (long) risc_state[4];
o = (long) risc_state[16] & 1;      /* the overflow bit */
```

This code is used in section 2.

8. ⟨ Use the RISC ELTE machine to compute the quotient and remainder, q and r 8 ⟩ ≡

```
memry[12] = div;
run_risc_elte(g, memry, memry_size, trace);
q = (long) risc_state[4];
r = ((long)(risc_state[2] + n)) & #7fff;
```

This code is used in section 2.

9. ⟨ Test some instructions of the RISC ELTE machine 9 ⟩ ≡

```
memry[0] = #f87;
memry[8] = m;
memry[10] = n;
memry[12] = test;
run_risc_elte(g, memry, memry_size, trace);
```

This code is used in section 2.

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

⟨ Global variables 3, 6 ⟩ Used in section 2.
⟨ Now do the same thing for n instead of m 5 ⟩ Used in section 4.
⟨ Prompt for two numbers; **break** if unsuccessful 4 ⟩ Used in section 2.
⟨ Test some instructions of the RISC ELTE machine 9 ⟩ Used in section 2.
⟨ Use the RISC ELTE machine to compute the product, p 7 ⟩ Used in section 2.
⟨ Use the RISC ELTE machine to compute the quotient and remainder, q and r 8 ⟩ Used in section 2.

December 2, 2013 at 10:06

TEST_RISC_ELTE

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