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

**1. Introduction.** This demonstration program uses graphs constructed by the *risc* procedure in the GB_GATES module to produce an interactive program called `take_risc`, 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 '`take_risc` ⟨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 machine to compute their product and quotient. 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.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(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.\n");
    while (1) {
      ⟨ Prompt for two numbers; break if unsuccessful 4 ⟩;
      ⟨ Use the RISC 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 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 $y \ge 0$ 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 7   /* location 'div' in the program below */
#define mult 10   /* location 'mult' in the program below */
#define memry_size 34   /* the number of instructions in the program below */
```

⟨ Global variables 3 ⟩ +≡

```
unsigned long memry[memry_size] = {   /* a "read-only memory" used by run_risc */
  #2ff0,   /* start: r2 = m (contents of next word) */
  #1111,   /* (we will put the value of m here, in memry[1]) */
  #1a30,   /*    r1 = 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 either mult or div here, in memry[5]) */
  #0f8f,   /* halt without changing any status bits */
  #3a21,   /* div: r3 = r1 */
  #1a01,   /*    r1 = 1 */
  #0a12,   /*    goto tri (literally, r0 += 2) */
  #3a01,   /* mult: r3 = 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) */
```

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

```
memry[1] = m;
memry[3] = n;
memry[5] = mult;
run_risc(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 machine to compute the quotient and remainder, $q$ and $r$ 8 ⟩ ≡

```
memry[5] = div;
run_risc(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. Index.** Finally, here's a list that shows where the identifiers of this program are defined and used.

May 19, 2018 at 02:30

# TAKE_RISC






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