Proof of a GCD Algorithm on an MC68020

R. S. Boyer
University of Texas at Austin
and Computational Logic, Inc.

Y. Yu
Digital Equipment Corporation

Adapted by P. B. Ladkin

This specification and proof is due to Robert S. Boyer and Yuan Yu, and is taken and adapted with permission by P. B. Ladkin from [BY92] and comments of R. S. Boyer.

References

[BM88] R.S. Boyer and J.S. Moore. *A Computational Logic Handbook*. Academic Press, 1988.

[BY92] R.S. Boyer and Yuan Yu. Automated correctness proofs of machine code programs for a commercial microprocessor. In D. Kapur, editor, *Automated Deduction - CADE11*, volume 607 of *Lecture Notes in Computer Science*, pages 416–430. Springer Verlag, 1992.

[Yu92] Yuan Yu. *Automated Proofs of Object Code for a Widely Used Microprocessor*. PhD thesis, University of Texas at Austin, Dec. 1992. Available from University Microfilms, Ann Arbor, Michigan.

[Yu93] Yuan Yu. Automated proofs of object code for a widely used microprocessor. Research Report 114, Digital Equipment Corporation Systems Research Center, Palo Alto, California, Oct. 5 1993.

The Specification and Proof

In the following, you will see a C program that computes GCD, followed by the binary for it printed by GDB when the GCC produced object program is loaded into a Sun-3, and a bunch of theorems, in the logic of Nqthm [BM88] about those bytes. The theorem below called gcd-correctness is the punch line, and it says that the bytes in question compute GCD on an MC68020.

The previous English sentence is only an approximation to what the theorem says; less informally, the theorem *gcd-correctness* below states that if `s` is an MC68020 state as characterized by `gcd-statep(s,a,b)`, then there is an integer `n`, given by the expression `gcd-t(a,b)` which tells us how many instructions to run the MC68020 starting with `s` before the GCD program returns, such that after running `s` for `n` steps the resulting state `s'` has these properties:

- `s'` is still "running" i.e., no errors occurred.
- The pc of `s'` points to the return address on the top of the stack in `s`.
- Register `D0` of `s'` contains `gcd(a,b)`, the greatest common divisor of `a` and `b`.
- Register `A6`, which is used by the `LINK` instruction, is unchanged.
- Register `A7`, the stack pointer, has been incremented by 4.

The theorem *gcd-correctness* further asserts that all of the registers of `s'` have the same values as those of `s`, except `D0`, `D1`, `A0`, `A1`, `A6`, and `A7`. Finally, the theorem *gcd-correctness* asserts that every memory location of `s'` has the same value as it did in `s`, except for the 12 bytes on either side of the stack pointer of `s`. (`read-rn` and `read-mem` are the primary functions for reading the contents of the registers and memory.)

All these statements are theorems mechanically checked by the theorem-prover. (There are other nice theorems, such as that the complexity of gcd is 580, i.e., the computation takes at most 580 instructions.) For reasons of brevity, the definition of the MC68020 architecture, which is about 100 pages of formal text, is omitted. This is all mainly the work of Yuan Yu, and may be obtained from him or from R.S. Boyer.

There are unstated conditions that we must apply before using the theorems mentioned. For example, the MC68020 spec referred to does not deal with interrupts (it could, that's just more work to do).

The rest of this appendix is *definitions* and *theorems*, in the formal logic Nqthm, mechanically verified.

```
;              Proof of the Correctness of a GCD Program
#|
```

The following C program computes the greatest common divisor of two integers a and b. We, here, investigate the machine code of this program generated by a widely used C compiler, and verify the correctness of the code.

```
/* computes the greatest common divisor by Euclid's algorithm */
gcd(int a, int b)
{
  while (a != 0){
    if (b == 0) return (a);
    if (a > b)
      a = a % b;
    else b = b % a;
  };
  return (b);
}
```

Here is the MC68020 assembly code of the above GCD program. The code is generated by "gcc -O".

```
0x22c6 <gcd>:           linkw a6,#0
0x22ca <gcd+4>:         moveml d2-d3,sp@-
0x22ce <gcd+8>:         movel a6@(8),d2
0x22d2 <gcd+12>:        movel a6@(12),d3
0x22d6 <gcd+16>:        tstl d2
0x22d8 <gcd+18>:        beq 0x22f6 <gcd+48>
0x22da <gcd+20>:        tstl d3
0x22dc <gcd+22>:        bne 0x22e2 <gcd+28>
0x22de <gcd+24>:        movel d2,d0
0x22e0 <gcd+26>:        bra 0x22f8 <gcd+50>
0x22e2 <gcd+28>:        cmpl d2,d3
0x22e4 <gcd+30>:        bge 0x22ee <gcd+40>
0x22e6 <gcd+32>:        divsll d3,d0,d2
0x22ea <gcd+36>:        movel d0,d2
0x22ec <gcd+38>:        bra 0x22d6 <gcd+16>
0x22ee <gcd+40>:        divsll d2,d0,d3
```

```
0x22f2 <gcd+44>:        movel d0,d3
0x22f4 <gcd+46>:        bra 0x22d6 <gcd+16>
0x22f6 <gcd+48>:        movel d3,d0
0x22f8 <gcd+50>:        moveml a6@(-8),d2-d3
0x22fe <gcd+56>:        unlk a6
0x2300 <gcd+58>:        rts

The machine code of the above program is:

<gcd>:       0x4e56  0x0000  0x48e7  0x3000  0x242e  0x0008  0x262e  0x000c
<gcd+16>:    0x4a82  0x671c  0x4a83  0x6604  0x2002  0x6016  0xb682  0x6c08
<gcd+32>:    0x4c43  0x2800  0x2400  0x60e8  0x4c42  0x3800  0x2600  0x60e0
<gcd+48>:    0x2003  0x4cee  0x000c  0xfff8  0x4e5e  0x4e75

'(78     86      0       0       72      231     48      0
  36     46      0       8       38      46      0       12
  74     130     103     28      74      131     102     4
  32     2       96      22      182     130     108     8
  76     67      40      0       36      0       96      232
  76     66      56      0       38      0       96      224
  32     3       76      238     0       12      255     248
  78     94      78      117)
|#

; now we start the correctness proof of this GCD program, defined by
; (gcd-code).
(defn gcd-code ()
  '(78     86      0       0       72      231     48      0
    36     46      0       8       38      46      0       12
    74     130     103     28      74      131     102     4
    32     2       96      22      182     130     108     8
    76     67      40      0       36      0       96      232
    76     66      56      0       38      0       96      224
    32     3       76      238     0       12      255     248
    78     94      78      117))
```

```
(constrain gcd-load (rewrite)
   (equal (gcd-loadp s)
          (and (evenp (gcd-addr))
               (numberp (gcd-addr))
               (nat-rangep (gcd-addr) 32)
               (rom-addrp (gcd-addr) (mc-mem s) 60)
               (mcode-addrp (gcd-addr) (mc-mem s) (gcd-code))))
    ((gcd-loadp (lambda (s) f))
     (gcd-addr (lambda () 1))))

(prove-lemma stepn-gcd-loadp (rewrite)
     (equal (gcd-loadp (stepn s n))
            (gcd-loadp s)))

; the functional description of the program (gcd-code).
(defn gcd (a b)
  (if (zerop a)
      (fix b)
    (if (zerop b)
        a
      (if (lessp b a)
          (gcd (remainder a b) b)
        (gcd a (remainder b a)))))
  ((lessp (plus a b))))

; the clock function.
(defn gcd-t1 (a b)
  (if (zerop a)
      6
      (if (zerop b)
          9
        (if (lessp b a)
            (splus 9 (gcd-t1 (remainder a b) b))
          (splus 9 (gcd-t1 a (remainder b a))))))
  ((lessp (plus a b))))
```

```
(defn gcd-t (a b)
  (splus 4 (gcd-t1 a b)))

; an induction hint.
(defn gcd-induct (s a b)
  (if (or (zerop a) (zerop b))
      t
    (if (lessp b a)
        (gcd-induct (stepn s 9) (remainder a b) b)
      (gcd-induct (stepn s 9) a (remainder b a))))
  ((lessp (plus a b))))

; the initial state.
(defn gcd-statep (s a b)
  (and (equal (mc-status s) 'running)
       (evenp (mc-pc s))
       (rom-addrp (mc-pc s) (mc-mem s) 60)
       (mcode-addrp (mc-pc s) (mc-mem s) (gcd-code))
       (ram-addrp (sub 32 12 (read-sp s)) (mc-mem s) 24)
       (equal a (iread-mem (add 32 (read-sp s) 4) (mc-mem s) 4))
       (equal b (iread-mem (add 32 (read-sp s) 8) (mc-mem s) 4))
       (numberp a)
       (numberp b)))
```

[Multiple intermediate lemmas here are omitted. See [BY92] and [Yu92, Yu93] for details.]

```
; the correctness of gcd.
(prove-lemma gcd-correctness (rewrite)
     (let ((sn (stepn s (gcd-t a b))))
       (implies (gcd-statep s a b)
(and (equal (mc-status sn) 'running)
     (equal (mc-pc sn) (rts-addr s))
     (equal (read-rn 32 14 (mc-rfile sn))
    (read-rn 32 14 (mc-rfile s)))
     (equal (read-rn 32 15 (mc-rfile sn))
```

```
   (add 32 (read-an 32 7 s) 4))
    (implies (and (d2-7a2-5p rn)
  (leq oplen 32))
     (equal (read-rn oplen rn (mc-rfile sn))
    (read-rn oplen rn (mc-rfile s))))
    (implies (disjoint x k (sub 32 12 (read-sp s)) 24)
     (equal (read-mem x (mc-mem sn) k)
    (read-mem x (mc-mem s) k)))
    (equal (iread-dn 32 0 sn) (gcd a b)))))
    ((disable gcd-statep gcd-s0p linked-rts-addr linked-a6)))

(disable gcd-t)

; to prove that (gcd a b) does compute the greatest common divisor, we need
; to prove the following two theorems:
;     1. (gcd a b) divides a and (gcd a b) divides b.
;        i.e. (gcd a b) is a common divisor of a and b.
;     2. any divisor of a and b is no greater than (gcd a b).
(prove-lemma remainder-remainder (rewrite)
     (implies (equal (remainder b c) 0)
      (equal (remainder (remainder a b) c)
     (remainder a c))))

(prove-lemma gcd-is-cd (rewrite)
     (and (equal (remainder a (gcd a b)) 0)
  (equal (remainder b (gcd a b)) 0)))

(prove-lemma gcd-the-greatest (rewrite)
     (implies (and (not (zerop a))
   (not (zerop b))
   (equal (remainder a x) 0)
   (equal (remainder b x) 0))
      (not (lessp (gcd a b) x)))
     ((induct (gcd a b))))

; a simple timing analysis.
```

```
(prove-lemma lessp-times-lessp (rewrite)
     (implies (and (lessp a b)
   (not (zerop c)))
      (lessp a (times b c))))

(prove-lemma remainder-shrink-half (rewrite)
     (implies (and (leq b a)
   (not (zerop b)))
      (not (lessp (quotient a 2) (remainder a b))))
     ((expand (times 2 x))
      (enable quotient-generalize)))

(prove-lemma gcd-t-shrink-1 (rewrite)
     (implies (and (leq b a)
   (not (zerop b)))
     (not (lessp (sub1 (log 2 a)) (log 2 (remainder a b)))))
     ((use (log-leq (b 2) (x (remainder a b)) (y (quotient a 2))))
      (disable quotient-times-lessp)))

(prove-lemma gcd-t-shrink-2 (rewrite)
     (implies (and (leq b a)
   (not (zerop b))
   (not (equal a 1)))
      (not (lessp (times 9 (plus x (log 2 a)))
  (plus 9 (times 9 (plus x (log 2 (remainder a b))))))))
     ((use (log-leq (b 2) (x (remainder a b)) (y (quotient a 2))))
      (disable quotient-times-lessp)))

(defn gcd-t2 (a b)
  (if (zerop a)
      6
    (if (zerop b)
9
      (if (lessp b a)
  (plus 9 (gcd-t2 (remainder a b) b))
(if (lessp a b)
```

```
    (plus 9 (gcd-t2 a (remainder b a)))
  18))))
  ((lessp (plus a b))))

(prove-lemma gcd-t2-ub ()
     (leq (gcd-t2 a b) (plus 18 (times 9 (plus (log 2 a) (log 2 b))))))

(prove-lemma gcd-t1-t2 (rewrite)
     (equal (gcd-t1 a b) (gcd-t2 a b))
     ((enable splus)))

(prove-lemma gcd-t-ub ()
     (leq (gcd-t a b)
  (plus 22 (times 9 (plus (log 2 a) (log 2 b)))))
     ((use (gcd-t2-ub))
      (enable gcd-t splus)))

(prove-lemma gcd-t-ubound ()
     (implies (and (lessp a (exp 2 31))
   (lessp b (exp 2 31)))
      (leq (gcd-t a b) 580))
     ((use (ta-lemma-2 (x 22) (y 9) (a1 (exp 2 31)) (b1 (exp 2 31)))
   (gcd-t-ub))))
```