Appendix A
Generic Theories for ACL2

```
;;; Loop Invariant Theory

;;; (wp s) = (if (b s) (qp s) (wp (sigma s)))

;;; (not (b s)) ^ (r s) => (r (sigma s))
;;; (b s) ^ (r s) => (qp s)

(encapsulate
 (((measure *) => *)
  ((sigma *) => *)
  ((b *) => *)
  ((qp *) => *)
  ((wp *) => *)
  ((r *) => *))
 (local (defun measure (s) (1+ (nfix (car s)))))
 (local (defun sigma (s) (cons (1- (car s)) (cdr s))))
 (local (defun b (s) (zp (car s))))
 (local (defun qp (s) (declare (ignore s)) t))
 (local (defun wp (s) (declare (ignore s)) t))
 (local (defun r (s) (declare (ignore s)) t))
 (defthm loop-invariant-wp-def-1
   (implies (b s) (equal (wp s) (qp s))))
 (defthm loop-invariant-wp-def-2
   (implies (and (not (b s)) (r s)) (equal (wp (sigma s)) (wp s))))
 (defthm loop-invariant-ordinalp (e0-ordinalp (measure s)))
 (defthm loop-invariant-ord-<
   (implies (not (b s)) (e0-ord-< (measure (sigma s)) (measure s))))
 (defthm loop-invariant-r-1
   (implies (and (not (b s)) (r s)) (r (sigma s))))
 (defthm loop-invariant-r-2
   (implies (and (b s) (r s)) (qp s))))

;;; An induct hint

(defun wp-ind (s)
  (declare (xargs :measure (measure s)))
  (if (b s) (qp s) (wp-ind (sigma s))))

;;; The main result derived from the loop-invariant theory.

(defthm wp-is-weakest-invariant
  (implies (r s) (wp s))
  :hints (("Goal"
            :induct (wp-ind s))))
```

;;; Tail Recursion Theory

```
;;; (g a s) = (if (bb s) (qt a s) (g (rho a s) (tau s)))
;;; (h s) = (if (bb s) (id) (rhoh (h (tau s)) s))
;;; (rt a s) is an invariant which assures desired properties of rho, op and (id)
;;; (hs s) = (if (bb s) s (hs (tau s))), computes a bottom object under tau

(encapsulate
 (((measure-g *) => *)
  ((tau *) => *)
  ((rt * *) => *)
  ((rho * *) => *)
  ((rhoh * *) => *)
  ((bb *) => *)
  ((qt * *) => *)
  ((g * *) => *)
  ((h *) => *)
  ((id) => *)
  ((op * * *) => *)
  ((hs *) => *))
 (local (defun measure-g (s) (1+ (nfix s))))
 (local (defun tau (s) (1- (nfix s))))
 (local (defun qt (a s) (declare (ignore a s)) t))
 (local (defun rho (a s) (declare (ignore a s)) 0))
 (local (defun rhoh (a s) (declare (ignore a s)) 0))
 (local (defun bb (s) (zp s)))
 (local (defun g (a s) (declare (ignore a s)) t))
 (local (defun id () 0))
 (local (defun op (a x s) (if (and (zp s) (equal x 0)) a 0)))
 (local (defun h (s) (declare (ignore s)) 0))
 (local (defun rt (a s) (and (integerp a) (integerp s) (<= 0 a) (<= 0 s))))
 (local (defun hs (s) (if (zp s) s 0)))
 (defthm tail-recursion-g-def-1
   (implies (and (bb s) (rt a s)) (equal (g a s) (qt a s))))
 (defthm tail-recursion-g-def-2
   (implies (and (not (bb s)) (rt a s)) (equal (g (rho a s) (tau s)) (g a s))))
 (defthm tail-recursion-ordinalp
   (e0-ordinalp (measure-g s)))
 (defthm tail-recursion-ord-<
   (implies (not (bb s))
            (e0-ord-< (measure-g (tau s)) (measure-g s))))
 (defthm tail-recursion-h-def-1
   (implies (and (bb s) (rt a s)) (equal (h s) (id))))
 (defthm tail-recursion-h-def-2
   (implies (and (not (bb s)) (rt a s)) (equal (rhoh (h (tau s)) s) (h s))))
 (defthm tail-recursion-rt
   (implies (and (not (bb s)) (rt a s))
                (rt (rho a s) (tau s))))
 (defthm tail-recursion-hs-1
   (implies (bb s) (equal (hs s) s)))
 (defthm tail-recursion-hs-2
   (implies (not (bb s)) (equal (hs (tau s)) (hs s))))
 (defthm tail-recursion-op-rho-rhoh  ; generated by proof of a-g-as-op-h
   (implies (and (not (bb s)) (rt a s))
                (equal (op (rho a s) (h (tau s)) (tau s))
                          (op a (rhoh (h (tau s)) s) s))))
 (defthm tail-recursion-op-id  ; generated by proof of a-g-as-op-h
   (implies (and (bb s) (rt a s))
                (equal (op a (id) s) a))))

(defun a-g (a s)
  (declare (xargs :measure (measure-g s)))
  (if (bb s)
      a
    (a-g (rho a s) (tau s))))

(defthm a-g-as-op-h
  (implies (rt a s)
           (equal (a-g a s)
                     (op a (h s) s)))
  :hints (("Goal"
            :induct (a-g a s))))

(defthm g-as-a-g
  (implies (rt a s)
              (equal (g a s) (qt (a-g a s) (hs s))))
  :hints (("Goal"
            :induct (a-g a s))))

;;; Main result of tail recursion theory

(defthm g=h
  (implies (rt a s)
              (equal (g a s)
                        (if (bb s)
                           (qt a s)
                         (qt (op a (h s) s) (hs s)))))
  :hints (("Goal"
             :induct (a-g a s))))
```

;;; Alternative Induction Theory

```
;;; The following theory can be used to prove equivalence, under the hypothesis
;;; p, between alternative representations fn1 and fn2 of the same function
;;; or in developing induction patterns different from that suggested by the
;;; standard definition.  The function id-alt below has dual roles.  When proving
;;; equivalence between fn1 and fn1, id-alt is the identity function.  Otherwise it
;;; represents the alternative induction.  In this latter case sigma1 = sigma2.
;;;
;;;      (fn1 s) = (if (b1 s) (q1 s) (fn1 (sigma1 s)))
;;;      (fn2 s) = (if (b2 s) (q2 s) (fn2 (sigma2 s)))

(encapsulate
 (((fn1 *) => *)
  ((fn2 *) => *)
  ((b1 *) => *)
  ((b2 *) => *)
  ((q1 *) => *)
  ((q2 *) => *)
  ((p *) => *)
  ((measure1 *) => *)
  ((sigma1 *) => *)
  ((sigma2 *) => *)
  ((id-alt *) => *))
 (local (defun fn1 (s) (declare (ignore s)) t))
 (local (defun fn2 (s) (declare (ignore s)) t))
 (local (defun b1 (s) (zp (car s))))
 (local (defun b2 (s) (zp (cdr s))))
 (local (defun q1 (s) (declare (ignore s)) t))
 (local (defun q2 (s) (declare (ignore s)) t))
 (local (defun p (s) (declare (ignore s)) t))
 (local (defun measure1 (s) (1+ (nfix (car s)))))
 (local (defun sigma1 (s) (cons (1- (nfix (car s))) (cdr s))))
 (local (defun sigma2 (s) (cons (car s) (1- (nfix (cdr s))))))
 (local (defun id-alt (s) (cons (cdr s) (car s))))
 (defthm alternative-induction-fn1-def-1
   (implies (b1 s) (equal (fn1 s) (q1 s))))
 (defthm alternative-induction-fn1-def-2
   (implies (and (not (b1 s)) (p s)) (equal (fn1 (sigma1 s)) (fn1 s))))
 (defthm alternative-induction-fn2-def-1
   (implies (b2 s) (equal (fn2 s) (q2 s))))
 (defthm alternative-induction-fn2-def-2
   (implies (not (b2 s)) (equal (fn2 (sigma2 s)) (fn2 s))))
 (defthm alternative-induction-ordinalp
   (e0-ordinalp (measure1 s)))
 (defthm alternative-induction-ord-<
   (implies (not (b1 s)) (e0-ord-< (measure1 (sigma1 s)) (measure1 s))))
 (defthm alternative-induction-id-alt
   (implies (and (p s) (not (b1 s)))
            (equal (id-alt (sigma1 s)) (sigma2 (id-alt s)))))
 (defthm alternative-induction-b2-id-alt
   (implies (p s) (equal (b2 (id-alt s)) (b1 s))))
 (defthm alternative-induction-p
   (implies (and (not (b1 s)) (p s)) (p (sigma1 s))))
 (defthm alternative-induction-q2-id-alt
   (implies (and (b1 s) (p s)) (equal (q2 (id-alt s)) (q1 s)))))

(defun fn1-ind (s)
  (declare (xargs :measure (measure1 s)))
  (if (b1 s) (q1 s) (fn1-ind (sigma1 s))))

;;; This is the main goal of the alternative induction theory.

(defthm fn1-as-fn2
  (implies (p s)
           (equal (fn1 s) (fn2 (id-alt s))))
  :hints (("Goal" :induct (fn1-ind s))))
```

Appendix B
Generic Theories for NQTHM

;;; Loop Invariant Theory

```
;;; (wp s) = (if (b s) (q s) (wp (sigma s)))
;;; (r s) = loop invariant

(constrain loop-invariant (rewrite)
  (and (implies (and (b s) (r s)) (equal (wp s) (qp s)))
       (implies (and (not (b s)) (r s)) (equal (wp (sigma s)) (wp s)))
       (ordinalp (measure s))
       (implies (not (b s)) (ord-lessp (measure (sigma s)) (measure s)))
       (implies (and (not (b s)) (r s)) (r (sigma s)))
       (implies (and (b s) (r s)) (qp s)))
  ((b (lambda (s) (zerop s)))
   (qp (lambda (s) t))
   (wp (lambda (s) t))
   (measure (lambda (s) (add1 s)))
   (sigma (lambda (s) (sub1 s)))
   (r (lambda (s) t))))

(defn wp-ind (s)
  (if (b s)
      t
    (wp-ind (sigma s)))
  ((ord-lessp (measure s))))

;;; Main result of the loop invariant theory

(prove-lemma wp-is-weakest-invariant (rewrite)
  (implies (r s) (wp s))
  ((induct (wp-ind s))))
```

;;; Tail Recursion Theory

```
;;; g is the tail recursive function.
;;; h is its primitive recursive counterpart.
;;; rho transforms the "a" component of state.
;;; tau transforms the "s" component of state.
;;; rhoh is the primitive recursive counterpart to rho.
;;; op computes the "a" state valued counterpart of g using h.
;;; id is an identity for op.

;;; (g a s) = (if (bb s) (qt a s) (g (rho a s) (tau s)))
;;; (h s) = (if (bb s) (id) (rhoh (h (tau s)) s))
;;; (rt a s) is an invariant which assures desired properties of rho, op and (id)
;;; (hs s) = (if (bb s) s (hs (tau s))), computes a bottom object under tau

(constrain tail-recursion-theory (rewrite)
 (and
  (implies (and (bb s) (rt a s)) (equal (g a s) (qt a s)))
  (implies (and (not (bb s)) (rt a s)) (equal (g (rho a s) (tau s)) (g a s)))
  (ordinalp (measure-g s))
  (implies (not (bb s)) (ord-lessp (measure-g (tau s)) (measure-g s)))
  (implies (and (bb s) (rt a s)) (equal (h s) (id)))
  (implies (and (not (bb s)) (rt a s)) (equal (rhoh (h (tau s)) s) (h s)))
  (implies (and (not (bb s)) (rt a s)) (rt (rho a s) (tau s)))
  (implies (bb s) (equal (hs s) s))
  (implies (not (bb s)) (equal (hs (tau s)) (hs s)))
  (implies (and (not (bb s)) (rt a s))
           (equal (op (rho a s) (h (tau s)) (tau s))
                  (op a (rhoh (h (tau s)) s) s)))
  (implies (and (bb s) (rt a s)) (equal (op a (id) s) a)))
 ((measure-g (lambda (s) (add1 s)))
  (tau (lambda (s) (sub1 s)))
  (qt (lambda (a s) t))
  (rho (lambda (a s) 0))
  (rhoh (lambda (a s) 0))
  (bb (lambda (s) (zerop s)))
  (g (lambda (a s) t))
  (id (lambda () 0))
  (op (lambda (a x s) (if (and (zerop s) (equal x 0)) a 0)))
  (h (lambda (s) 0))
  (rt (lambda (a s) t))
  (hs (lambda (s) (if (zerop s) s 0)))))

(defn a-g (a s)
  (if (bb s)
      a
    (a-g (rho a s) (tau s)))
  ((ord-lessp (measure-g s))))

(prove-lemma a-g-as-op-h (rewrite)
  (implies (rt a s)
             (equal (a-g a s)
                    (op a (h s) s))))

(prove-lemma g-as-a-g (rewrite)
  (implies (rt a s)
           (equal (g a s) (qt (a-g a s) (hs s))))
  ((induct (a-g a s))))
```

```
;;; Main result of tail recursion theory

(prove-lemma g=h (rewrite)
  (implies (rt a s)
             (equal (g a s)
                    (if (bb s)
                       (qt a s)
                     (qt (op a (h s) s) (hs s))))))
```

;;; Alternative Induction Theory

```
(constrain alternate-induction-theory (rewrite)
 (and (implies (and (p s) (not (b1 s))) (equal (id-alt (sigma1 s)) (sigma2 (id-alt s))))
       (implies (not (b1 s)) (equal (fn1 (sigma1 s)) (fn1 s)))
       (implies (b2 s) (equal (fn2 s) (q2 s)))
       (implies (b1 s) (equal (fn1 s) (q1 s)))
       (implies (not (b1 s)) (ord-lessp (measure1 (sigma1 s)) (measure1 s)))
       (implies (not (b2 s)) (equal (fn2 (sigma2 s)) (fn2 s)))
       (implies (p s) (equal (b2 (id-alt s)) (b1 s)))
       (implies (and (not (b1 s)) (p s)) (p (sigma1 s)))
       (implies (and (b1 s) (p s)) (equal (q2 (id-alt s)) (q1 s)))
       (ordinalp (measure1 s)))
 ((fn1 (lambda (s) t))
  (fn2 (lambda (s) t))
  (b1 (lambda (s) (zerop (car s))))
  (b2 (lambda (s) (zerop (cdr s))))
  (q1 (lambda (s) t))
  (q2 (lambda (s) t))
  (p (lambda (s) t))
  (measure1 (lambda (s) (add1 (car s))))
  (sigma1 (lambda (s) (cons (sub1 (car s)) (cdr s))))
  (sigma2 (lambda (s) (cons (car s) (sub1 (cdr s)))))
  (id-alt (lambda (s) (cons (cdr s) (car s))))))

(defn fn1-ind (s)
  (if (b1 s) (q1 s) (fn1-ind (sigma1 s)))
  ((ord-lessp (measure1 s))))

;;; This is the main result of the theory.

(prove-lemma fn1-as-fn2 nil
            (implies (p s)
                     (equal (fn1 s) (fn2 (id-alt s))))
            ((induct (fn1-ind s))))
```

Appendix C

The Sum Program - ACL2

```
;;; The following program sums the integers from 1 to N on the Mostek
;;; 6502 microprocessor.  A is an 8-bit wide accumulator. N is a single byte
;;; of data in memory.  C is a carry flag.  We prove that the limited (8-bit)
;;; precision computation delivers the correct result, provided N*(N+1)/2
;;; is less than 256 and N is greater than 0.

;;;          LDA #0      ; load A immediate with the constant 0
;;;          CLC         ; clear the carry flag
;;; LOOP  ADC N        ; add with carry N to A
;;;          DEC N        ; decrement N
;;;          BNE LOOP   ; branch if N is non-zero to LOOP

;;; Provide semantics for the 6502 DEC instruction.

(defun dec (x)
  (if (zp x)
      255
    (1- x)))

;;; Mechanically derived

(defun wp-loop (n a c nsave)
  (declare (xargs :measure (dec n)))
  (if (equal (dec n) 0)
      (equal (mod (+ c (+ a n)) 256)
             (floor (* nsave (1+ nsave)) 2))
    (wp-loop (dec n)
               (mod (+ c (+ a n)) 256)
               (floor (+ c (+ a n)) 256)
               nsave)))

;;; Weakest precondition at beginning of program

(defun wp-1 (n nsave)
  (wp-loop n 0 0 nsave))

(defmacro natp (x) ‘(and (integerp ,x) (<= 0 ,x)))
```

;;; Proof by Generalization

```
;;; This compensates for an error within arithmetic-4

(defthm equal-transpose-constant
  (equal (equal (+ -1 a) 0)
            (equal a 1)))

(defthm wp-sum-loop-generalization
  (implies (and (not (zp n))
                    (< (+ a (floor (* n (1+ n)) 2)) 256)
                    (natp a)
                    (equal c 0)
                    (natp nsave))
              (equal (wp-loop n a c nsave)
                     (equal (+ a (floor (* n (1+ n)) 2))
                            (floor (* nsave (1+ nsave)) 2))))))

(defthm wp-loop-is-correct
 (implies (and (not (zp n))
                   (equal nsave n)
                   (< (floor (* n (1+ n)) 2) 256))
            (wp-1 n nsave)))
```

;;; Proof Using the Loop Invariant Theory

```
(defun n (s) (car s))
(defun a (s) (cadr s))
(defun c (s) (caddr s))
(defun nsave (s) (cadddr s))

;;; This compensates for an error within arithmetic-4

(defthm equal-transpose-constant
  (equal (equal (+ -1 a) 0)
          (equal a 1)))

(defthm wp-sum-loop-invariant
 (implies (and (not (zp (n s)))
                   (< (+ (a s) (floor (* (n s) (1+ (n s))) 2)) 256)
                   (natp (a s))
                   (equal (c s) 0)
                   (natp (nsave s))
                   (equal (+ (a s) (floor (* (n s) (1+ (n s))) 2))
                          (floor (* (nsave s) (1+ (nsave s))) 2)))
              (wp-loop (n s) (a s) (c s) (nsave s)))
 :hints
 (("Goal"
   :use
   ((:functional-instance
     wp-is-weakest-invariant
     (b (lambda (s) (equal (dec (n s)) 0)))
     (qp (lambda (s) (equal (mod (+ (c s) (+ (a s) (n s))) 256)
                             (floor (* (nsave s) (1+ (nsave s))) 2))))
     (wp (lambda (s) (wp-loop (n s) (a s) (c s) (nsave s))))
     (measure (lambda (s) (dec (n s))))
     (sigma (lambda (s) (list (dec (n s))
                              (mod (+ (c s) (+ (a s) (n s))) 256)
                              (floor (+ (c s) (+ (a s) (n s))) 256)
                              (nsave s))))
     (r (lambda (s) (and (not (zp (n s)))
                          (< (+ (a s) (floor (* (n s) (1+ (n s))) 2)) 256)
                          (natp (a s))
                          (equal (c s) 0)
                          (natp (nsave s))
                          (equal (+ (a s) (floor (* (n s) (1+ (n s))) 2))
                                 (floor (* (nsave s) (1+ (nsave s))) 2))))))))))

(defthm wp-sum-loop-invariant-flat
 (implies (and (not (zp n))
                (< (+ a (floor (* n (1+ n)) 2)) 256)
                (natp a)
                (equal c 0)
                (natp nsave)
                (equal (+ a (floor (* n (1+ n)) 2))
                       (floor (* nsave (1+ nsave)) 2)))
           (wp-loop n a c nsave))
 :hints
 (("Goal"
   :in-theory (disable wp-sum-loop-invariant)
   :use (:instance
         wp-sum-loop-invariant
         (s (list n a c nsave))))))

(defthm wp-loop-is-correct
 (implies (and (not (zp n))
                (equal nsave n)
                (< (floor (* n (1+ n)) 2) 256))
          (wp-1 n nsave)))
```

;;; Proof Using the Tail Recursion Theory

```
;;; Represent the "a" component directly and "s" by a list.

(defun n (s) (car s))
(defun c (s) (cadr s))
(defun nsave (s) (caddr s))

;;; Define the instantiation of h from the generic theory.
```

```
(defun wp-loop-h (s)
  (declare (xargs :measure (dec (n s))))
  (if (equal (dec (n s)) 0)
      0
    (+ (n s) (wp-loop-h (list (dec (n s)) (c s) (nsave s))))))

;;;; Instantiate the generic theory

(defthm wp-loop-g=h
  (implies (and (not (zp (n s)))
                      (natp (nsave s))
                      (natp a)
                      (equal (c s) 0)
                      (< (+ a (floor (* (n s) (+ 1 (n s))) 2)) 256))
               (equal (wp-loop (n s) a (c s) (nsave s))
                         (if (equal (dec (n s)) 0)
                             (equal (mod (+ a (n s)) 256)
                                       (floor (* (nsave s) (+ 1 (nsave s))) 2))
                           (let ((a (+ a (wp-loop-h s)))
                                 (s (list 1 (c s) (nsave s))))
                             (equal (mod (+ a (n s)) 256)
                                       (floor (* (nsave s) (+ 1 (nsave s))) 2))))))
  :hints
  (("Goal"
    :use
    ((:functional-instance
      g=h
      (bb (lambda (s) (equal (dec (n s)) 0)))
      (qt (lambda (a s) (equal (mod (+ a (n s)) 256)
                                         (floor (* (nsave s) (+ 1 (nsave s))) 2))))
      (g (lambda (a s) (wp-loop (n s) a (c s) (nsave s))))
      (measure-g (lambda (s) (dec (n s))))
      (tau (lambda (s) (list (dec (n s)) (c s) (nsave s))))
      (rho (lambda (a s) (mod (+ a (c s) (n s)) 256)))
      (rhoh (lambda (a s) (+ a (n s))))
      (h (lambda (s) (wp-loop-h s)))
      (rt (lambda (a s) (and (not (zp (n s)))
                                    (natp (nsave s))
                                    (natp a)
                                    (equal (c s) 0)
                                    (< (+ a (floor (* (n s) (+ 1 (n s))) 2)) 256))))
      (id (lambda () 0))
      (op (lambda (a x s) (if (equal (dec (n s)) 0)
                                      a
                                    (+ a x))))
      (hs (lambda (s) (if (equal (dec (n s)) 0)
                       s
                     (list 1 (c s) (nsave s))))))))))

(defthm wp-loop-g=h-flat
  (implies (and (not (zp n))
                (natp nsave)
                (natp a)
                (equal c 0)
                (< (+ a (floor (* n (+ 1 n)) 2)) 256))
           (equal (wp-loop n a c nsave)
                  (if (equal (dec n) 0)
                      (equal (mod (+ a n) 256)
                             (floor (* nsave (+ 1 nsave)) 2))
                    (equal (mod (+ 1 a (wp-loop-h (list n c nsave))) 256)
                           (floor (* nsave (+ 1 nsave)) 2)))))
  :hints (("Goal"
           :use (:instance wp-loop-g=h
                           (a a)
                           (s (list n c nsave))))))

;;; Compensates for bug in arithmetic-4

(defthm equal-transpose-constant
  (equal (equal (+ -1 a) 0)
         (equal a 1)))

(defthm wp-loop-h-closed
  (implies (not (zp (n s)))
           (equal (wp-loop-h s)
                  (+ -1 (floor (* (n s) (+ 1 (n s))) 2)))))

(defthm wp-loop-is-correct
  (implies (and (not (zp n))
                (equal nsave n)
                (< (floor (* n (+ 1 n)) 2) 256))
           (wp-1 n nsave)))
```

;;; Proof Using the Alternative Induction Theory

```
(defun n (s) (car s))
(defun a (s) (cadr s))
(defun c (s) (caddr s))
(defun nsave (s) (cadddr s))

;;; Instantiate the theory for the alternative induction that decrements
;;; NSAVE by 1 and A by NSAVE.  This choice is motivated by leaving q1
;;; invariant and commuting with sigma1.
```

```
(defthm wp-loop-fn1-as-fn2
  (implies (and (not (zp (n s)))
                      (not (zp (nsave s)))
                      (equal (c s) 0)
                      (< (+ (a s) (floor (* (n s) (+ 1 (n s))) 2)) 256)
                      (natp (a s))
                      (<= (nsave s) (a s)))
               (equal (wp-loop (n s) (a s) (c s) (nsave s))
                         (wp-loop (n s)
                                      (- (a s) (nsave s))
                                      (c s)
                                      (+ -1 (nsave s))))))
  :hints
  (("Goal"
    :use
    (:functional-instance
     fn1-as-fn2
     (fn1 (lambda (s) (wp-loop (n s) (a s) (c s) (nsave s))))
     (fn2 (lambda (s) (wp-loop (n s) (a s) (c s) (nsave s))))
     (b1 (lambda (s) (equal (dec (n s)) 0)))
     (b2 (lambda (s) (equal (dec (n s)) 0)))
     (q1 (lambda (s) (equal (mod (+ (c s) (a s) (n s)) 256)
                                 (floor (* (nsave s) (+ 1 (nsave s))) 2))))
     (q2 (lambda (s) (equal (mod (+ (c s) (a s) (n s)) 256)
                                 (floor (* (nsave s) (+ 1 (nsave s))) 2))))
     (sigma1 (lambda (s)
              (list (dec (n s))
                    (mod (+ (c s) (a s) (n s)) 256)
                    (floor (+ (c s) (a s) (n s)) 256)
                    (nsave s))))
     (sigma2 (lambda (s)
              (list (dec (n s))
                    (mod (+ (c s) (a s) (n s)) 256)
                    (floor (+ (c s) (a s) (n s)) 256)
                    (nsave s))))
     (p (lambda (s)
          (and (not (zp (n s)))
                 (not (zp (nsave s)))
                 (< (+ (a s) (floor (* (n s) (+ 1 (n s))) 2)) 256)
                 (natp (a s))
                 (<= (nsave s) (a s))
                 (equal (c s) 0))))
     (id-alt (lambda (s)
              (list (n s)
                    (- (a s) (nsave s))
                    (c s)
                    (+ -1 (nsave s)))))
     (measure1 (lambda (s) (if (zp (n s)) 256 (n s))))))))

;;; Convert the above to an effective rewrite rule.

(defthm wp-loop-fn1-as-fn2-rewrite
  (implies (and (not (zp n))
                      (not (zp nsave))
                      (equal c 0)
                      (< (+ a (floor (* n (+ 1 n)) 2)) 256)
                      (natp a)
                      (<= nsave a))
              (equal (wp-loop n a c nsave)
                        (wp-loop n (- a nsave) c (+ -1 nsave))))
  :hints
  (("Goal"
    :use (:instance wp-loop-fn1-as-fn2
                    (s (list n a c nsave))))))

;;; Compensates for bug in arithmetic-4

(defthm equal-transpose-constant
  (equal (equal (+ -1 a) 0)
            (equal a 1)))

;;; Finally, the correctness result.

(defthm wp-loop-is-correct
  (implies (and (< (floor (* n (+ 1 n)) 2) 256)
                      (not (zp n))
                      (equal nsave n))
               (wp-1 n nsave)))
```

Appendix D

The Sum Program - NQTHM

```
;;; Provide semantics for the Mostek 6502 DEC instruction.  The remaining
;;; instructions have semantics built into the weakest precondition generation
;;; program.

(defn dec (x)
  (if (zerop x)
      255
    (sub1 x)))

;;; Mechanically generated

(DEFN WP-LOOP (N A C NSAVE)
  (IF (EQUAL (DEC N) 0)
      (EQUAL (REMAINDER (PLUS C (PLUS A N)) 256)
             (QUOTIENT (TIMES NSAVE (PLUS 1 NSAVE)) 2))
    (WP-LOOP (DEC N)
             (REMAINDER (PLUS C (PLUS A N)) 256)
             (QUOTIENT (PLUS C (PLUS A N)) 256)
             NSAVE))
  ((lessp (if (zerop n) 256 n))))

;;; Weakest precondition at start of program

(DEFN WP-1 (N NSAVE)
   (WP-LOOP N 0 0 NSAVE))

;;; Proof by generalization

(prove-lemma wp-loop-closed (rewrite)
   (implies (and (not (zerop n))
                 (lessp (plus c a (quotient (times n (add1 n)) 2))
                        256))
            (equal (wp-loop n a c nsave)
                   (equal (plus c a (quotient (times n (add1 n)) 2))
                          (quotient (times nsave (add1 nsave)) 2))))
   ((disable-theory if-normalization)
    (induct (wp-loop n a c nsave))
    (hands-off difference)
    (expand (times n (add1 n)))
    (disable quotient-add1-arg2
             remainder-add1-arg2
             remainder-plus-arg1
             remainder-plus-arg1-simple
```

```
                quotient-plus-arg1
                equal-add1
                quotient-times-arg1
                quotient-add1-arg1
                lessp-quotient
                lessp-quotient-arg2-linear
                times-add1
                sub1-plus)))

;;; Final correctness result

(prove-lemma wp-loop-is-correct (rewrite)
  (implies (and (not (zerop n))
                    (lessp (quotient (times n (add1 n)) 2) 256)
                    (equal nsave n))
               (wp-1 n nsave))
  ((hands-off quotient)))
```

;;; Proof Using the Loop Invariant Theory

```
(defn n (s) (car s))
(defn a (s) (cadr s))
(defn c (s) (caddr s))
(defn nsave (s) (cadddr s))

;;; Instantiate loop invariant theory

(functionally-instantiate wp-sum-loop-invariant nil
 (implies (and (not (zerop (n s)))
                   (lessp (plus (a s) (quotient (times (n s) (add1 (n s))) 2)) 256)
                   (equal (c s) 0)
                   (equal (plus (a s) (quotient (times (n s) (add1 (n s))) 2))
                          (quotient (times (nsave s) (add1 (nsave s))) 2)))
              (wp-loop (n s) (a s) (c s) (nsave s)))
 wp-is-weakest-invariant
 ((b (lambda (s) (equal (dec (n s)) 0)))
  (qp (lambda (s) (equal (remainder (plus (c s) (plus (a s) (n s))) 256)
                           (quotient (times (nsave s) (add1 (nsave s))) 2))))
  (wp (lambda (s) (wp-loop (n s) (a s) (c s) (nsave s))))
  (measure (lambda (s) (dec (n s))))
  (sigma (lambda (s) (list (dec (n s))
                            (remainder (plus (c s) (plus (a s) (n s))) 256)
                            (quotient (plus (c s) (plus (a s) (n s))) 256)
                            (nsave s))))
  (r (lambda (s) (and (not (zerop (n s)))
                       (lessp (plus (a s) (quotient (times (n s) (add1 (n s))) 2)) 256)
                       (equal (c s) 0)
```

```
                     (equal (plus (a s) (quotient (times (n s) (add1 (n s))) 2))
                            (quotient (times (nsave s) (add1 (nsave s))) 2))))))
 ((hands-off difference)
  (disable quotient-add1-arg2
             remainder-add1-arg2
             quotient-plus-arg1
             remainder-plus-arg1
             quotient-times-arg1
             quotient-times-arg1-kb
             remainder-difference-arg1
             lessp-quotient-arg2-linear
             quotient-remainder
             lessp-quotient)))

;;; Transform to flat state space rule.

(prove-lemma wp-sum-loop-invariant-flat (rewrite)
  (implies (and (not (zerop n))
                    (lessp (plus a (quotient (times n (add1 n)) 2)) 256)
                    (equal c 0)
                    (equal (plus a (quotient (times n (add1 n)) 2))
                           (quotient (times nsave (add1 nsave)) 2)))
             (wp-loop n a c nsave))
 ((use (wp-sum-loop-invariant (s (list n a c nsave))))
  (hands-off quotient remainder)))

;;; Final correctness result

(prove-lemma wp-loop-is-correct (rewrite)
   (implies (and (not (zerop n))
                     (lessp (quotient (times n (add1 n)) 2) 256)
                     (equal nsave n))
              (wp-1 n nsave))
   ((hands-off quotient)))
```

;;; Proof Using the Tail Recursion Theory

```
;;; Represent the "a" component of state directly by a, and the "s" component by a list.

(defn n (s) (car s))
(defn c (s) (cadr s))
(defn nsave (s) (caddr s))

;;; Define the instantiation of h from the generic theory.

(defn wp-loop-h (s)
  (if (equal (dec (n s)) 0)
    0
   (plus (n s) (wp-loop-h (list (dec (n s)) (c s) (nsave s)))))
  ((lessp (if (zerop (n s)) 256 (n s)))))

;;; Instantiate g=h from the tail recursion theory.

(functionally-instantiate wp-loop-g=h nil
  (implies (and (not (zerop (n s)))
                    (equal (c s) 0)
                    (lessp (plus a (quotient (times (n s) (add1 (n s))) 2)) 256))
               (equal (wp-loop (n s) a (c s) (nsave s))
                     (if (equal (dec (n s)) 0)
                        (equal (remainder (plus a (n s)) 256)
                               (quotient (times (nsave s) (add1 (nsave s))) 2))
                       (let ((a (if (equal (dec (n s)) 0)
                                    a
                                   (plus a (wp-loop-h s))))
                             (s (if (equal (dec (n s)) 0)
                                    s
                                   (list 1 (c s) (nsave s)))))
                         (equal (remainder (plus a (n s)) 256)
                                (quotient (times (nsave s) (add1 (nsave s))) 2))))))
  g=h
  ((bb (lambda (s) (equal (dec (n s)) 0)))
   (qt (lambda (a s) (equal (remainder (plus a (n s)) 256)
                            (quotient (times (nsave s) (add1 (nsave s))) 2))))
   (g (lambda (a s) (wp-loop (n s) a (c s) (nsave s))))
   (measure-g (lambda (s) (if (zerop (n s)) 256 (n s))))
   (tau (lambda (s) (list (dec (n s)) (c s) (nsave s))))
   (rho (lambda (a s) (remainder (plus a (c s) (n s)) 256)))
   (rhoh (lambda (a s) (plus a (n s))))
   (h (lambda (s) (wp-loop-h s)))
   (rt (lambda (a s) (and (not (zerop (n s)))
                          (equal (c s) 0)
                          (lessp (plus a (quotient (times (n s) (add1 (n s))) 2))
                                 256))))
 (id (lambda () 0))
   (op (lambda (a x s) (if (equal (dec (n s)) 0)
                           a
                          (plus a x))))
   (hs (lambda (s) (if (equal (dec (n s)) 0)
                       s
                      (list 1 (c s) (nsave s))))))
  ((disable-theory if-normalization)
   (expand (times v v) (times v w))
   (disable quotient-add1-arg2
            remainder-add1-arg2
```

```
                quotient-plus-arg1
                remainder-plus-arg1
                remainder-plus-arg1-simple
                equal-add1
                remainder-difference-arg1
                difference-add1
                times-add1
                difference-plus-arg1
                sub1-quotient
                sub1-remainder
                sub1-plus
                sub1-times
                dichotomy)))

;;; Transform above to a flat state space rule.

(prove-lemma wp-loop-g=h-rewrite (rewrite)
  (implies (and (not (zerop n))
                (equal c 0)
                (lessp (plus a (quotient (times n (add1 n)) 2)) 256))
           (equal (wp-loop n a c nsave)
                  (if (equal (dec n) 0)
                      (equal (remainder (plus a n) 256)
                             (quotient (times nsave (add1 nsave)) 2))
                    (let ((a (plus a (wp-loop-h (list n c nsave)))))
                      (equal (remainder (add1 a) 256)
                             (quotient (times nsave (add1 nsave)) 2))))))
 ((disable-theory if-normalization)
  (use (wp-loop-g=h (a a) (s (list n c nsave))))
  (hands-off difference)
  (disable quotient-add1-arg2
           remainder-add1-arg2
           quotient-plus-arg1
           remainder-plus-arg1
           remainder-plus-arg1-simple
           equal-add1
           times-add1
           plus-add1
           sub1-plus
           remainder-difference-arg1)))

;;; Closed form for wp-loop-h

(prove-lemma wp-loop-h-closed (rewrite)
  (implies (not (zerop (n s)))
                (equal (wp-loop-h s)
                       (sub1 (quotient (times (n s) (add1 (n s))) 2))))
  ((induct (wp-loop-h s))
   (disable quotient-add1-arg2
                remainder-add1-arg2
                sub1-quotient
                quotient-times-arg1
                quotient-times-arg1-kb
                quotient-plus-arg1
                remainder-difference-arg1)))

;;; Final correctness result

(prove-lemma wp-loop-is-correct (rewrite)
    (implies (and (not (zerop n))
                  (lessp (quotient (times n (add1 n)) 2) 256)
                  (equal nsave n))
             (wp-1 n nsave))
    ((disable-theory if-normalization)
     (hands-off difference)
     (disable quotient-add1-arg2
              remainder-add1-arg2
              quotient-plus-arg1
              remainder-plus-arg1
              remainder-plus-arg1-simple
              equal-add1
              times-add1
              sub1-plus
              plus-add1
              remainder-add1-arg1
              sub1-times)))
```

;;; Proof Using the Alternative Induction Theory

```
(defn n (s) (car s))
(defn a (s) (cadr s))
(defn c (s) (caddr s))
(defn nsave (s) (cadddr s))

;;; Instantiate the theory for the alternative induction that decrements
;;; NSAVE by 1 and A by NSAVE.  This choice is motivated by leaving q1
;;; invariant and commuting with sigma1.

(functionally-instantiate wp-loop-fn1-as-fn2 nil
  (implies (and (not (zerop (n s)))
                (not (zerop (nsave s)))
                (equal (c s) 0)
                (lessp (plus (a s) (quotient (times (n s) (add1 (n s))) 2)) 256)
                (not (lessp (a s) (nsave s))))
```

```
            (equal (wp-loop (n s) (a s) (c s) (nsave s))
                   (let ((s (list (n s)
                                  (difference (a s) (nsave s))
                                  (c s)
                                  (sub1 (nsave s)))))
                     (wp-loop (n s) (a s) (c s) (nsave s)))))
fn1-as-fn2
((fn1 (lambda (s) (wp-loop (n s) (a s) (c s) (nsave s))))
 (fn2 (lambda (s) (wp-loop (n s) (a s) (c s) (nsave s))))
 (b1 (lambda (s) (equal (dec (n s)) 0)))
 (b2 (lambda (s) (equal (dec (n s)) 0)))
 (q1 (lambda (s) (equal (remainder (plus (c s) (a s) (n s)) 256)
                        (quotient (times (nsave s) (add1 (nsave s))) 2))))
 (q2 (lambda (s) (equal (remainder (plus (c s) (a s) (n s)) 256)
                        (quotient (times (nsave s) (add1 (nsave s))) 2))))
 (sigma1 (lambda (s)
           (list (dec (n s))
                 (remainder (plus (c s) (a s) (n s)) 256)
                 (quotient (plus (c s) (a s) (n s)) 256)
                 (nsave s))))
 (sigma2 (lambda (s)
           (list (dec (n s))
                 (remainder (plus (c s) (a s) (n s)) 256)
                 (quotient (plus (c s) (a s) (n s)) 256)
                 (nsave s))))
 (p (lambda (s)
      (and (not (zerop (n s)))
           (not (zerop (nsave s)))
           (equal (c s) 0)
           (lessp (plus (a s) (quotient (times (n s) (add1 (n s))) 2)) 256)
           (not (lessp (a s) (nsave s))))))
 (id-alt (lambda (s)
           (list (n s)
                 (difference (a s) (nsave s))
                 (c s)
                 (sub1 (nsave s)))))
 (measure1 (lambda (s) (if (zerop (n s)) 256 (n s)))))
((disable-theory if-normalization)
 (expand (times (car s) (car s)))
 (disable quotient-add1-arg2
          remainder-add1-arg2
          quotient-plus-arg1
          remainder-plus-arg1
          remainder-plus-arg1-simple
          equal-add1
          plus-add1
          quotient-times-arg1
```

```
                quotient-remainder
                lessp-quotient-arg2-linear
                difference-add1
                difference-plus-arg1
                sub1-quotient
                lessp-times-single-linear
                sub1-plus
                lessp-odometer-simple
                equal-odometer-simple
                remainder-lessp-linear)))

;;; Convert the above to an effective rewrite rule.

(prove-lemma wp-loop-fn1-as-fn2-rewrite (rewrite)
  (implies (and (not (zerop n))
                      (not (zerop nsave))
                      (equal c 0)
                      (lessp (plus a (quotient (times n (add1 n)) 2)) 256)
                      (not (lessp a nsave)))
               (equal (wp-loop n a c nsave)
                         (wp-loop n (difference a nsave) c (sub1 nsave))))
  ((disable-theory if-normalization)
   (hands-off quotient remainder)
   (use (wp-loop-fn1-as-fn2 (s (list n a c nsave))))))

;;; The correctness result

(prove-lemma wp-loop-is-correct (rewrite)
    (implies (and (not (zerop n))
                       (lessp (quotient (times n (add1 n)) 2) 256)
                       (equal nsave n))
                (wp-1 n nsave))
    ((disable-theory if-normalization)
      (induct (difference n nsave))
      (disable quotient-plus-arg1
                  difference-add1
                  equal-add1
                  quotient-add1-arg2
                  remainder-add1-arg2
                  lessp-transpose-meta
                  equal-transpose-meta
                  remainder-plus-arg1
                  difference-plus-arg1
                  lessp-times-single-linear
                  remainder-lessp-linear
                  equal-odometer-simple
                  lessp-odometer-simple)))
```

Appendix E
The Multiply Program - ACL2

```
;;; The following is a Floyd-Hoare correctness specification for the multiply
;;; program.
;;;
;;;     { F1=F1SAVE ^ F1<256 ^ F2<256 ^ LOW<256 }
;;;
;;;             LDX #8        load the X register immediate with the value 8
;;;             LDA #0        load the A register immediate with the value 0
;;; LOOP        ROR F1        rotate F1 right circular through the carry flag
;;;             BCC ZCOEF     branch on carry flag clear to ZCOEF
;;;             CLC           clear the carry flag
;;;             ADC F2        add with carry F2 to the contents of A
;;; ZCOEF       ROR A         rotate A right circular through the carry flag
;;;             ROR LOW       rotate LOW right circular through the carry flag
;;;             DEX           decrement the X register by 1
;;;             BNE LOOP      branch if X is non-zero to LOOP
;;;
;;;     { LOW + 256*A = F1SAVE*F2 }

;;; Provide semantics for the Mostek 6502 DEX instruction.  The semantics of
;;; the remaining functions are built into the weakest precondition generation
;;; program.

(defun dec (x)
  (if (zp x)
      255
    (1- x)))

;;; This is mechanically derived.

(defun wp-zcoef (f1 x c low a f1save f2)
  (declare (xargs :measure (dec x)))
  (if (equal (dec x) 0)
      (equal (+ (* (+ (* 128 c) (floor a 2)) 256)
                (+ (* 128 (mod a 2))
                   (floor low 2)))
             (* f1save f2))
    (wp-zcoef (+ (* 128 (mod low 2)) (floor f1 2))
                (dec x)
                (* (mod f1 2) (floor (+ (+ (* 128 c) (floor a 2)) f2) 256))
                (+ (* 128 (mod a 2)) (floor low 2))
                (if (equal (mod f1 2) 0)
                    (+ (* 128 c) (floor a 2))
                  (mod (+ (+ (* 128 c) (floor a 2)) f2) 256))
```

```
                    f1save
                    f2)))
```

;;; This is the weakest precondition at the beginning of the program (a slight
;;; editing of the mechanically derived formula).

```
(defun wp-zcoef-1 (f1 c low f2)
  (wp-zcoef  (+ (* 128 c) (floor f1 2))
                   8
                   0
                   low
                   (* (mod f1 2) f2)
                   f1
                   f2))
```

;;; We generalize the register size in order to capture properties of the constants
;;; 128 and 256.

```
(defun wp-zcoef-g (f1 x c low a result f2 i)
  (declare (xargs :measure (dec x)))
  (if (equal (dec x) 0)
      (equal (+ (* (+ (* (expt 2 (1- i)) c) (floor a 2)) (expt 2 i))
                   (+ (* (expt 2 (1- i)) (mod a 2))
                       (floor low 2)))
                 result)
    (wp-zcoef-g (+ (* (expt 2 (1- i)) (mod low 2))
                          (floor f1 2))
                      (dec x)
                      (* (mod f1 2)
                          (floor (+ (+ (* (expt 2 (1- i)) c) (floor a 2)) f2)
                                 (expt 2 i)))
                      (+ (* (expt 2 (1- i)) (mod a 2))
                          (floor low 2))
                      (if (equal (mod f1 2) 0)
                          (+ (* (expt 2 (1- i)) c) (floor a 2))
                        (mod (+ (+ (* (expt 2 (1- i)) c) (floor a 2)) f2)
                               (expt 2 i)))
                      result
                      f2
                      i)))
```

```
(defmacro natp (x) ‘(and (integerp ,x) (<= 0 ,x)))
```

;;; Proof by Generalization

;;; Our proof strategy is to successively transform wp-zcoef-1 into instances of more
;;; general functions. This performs the first step.

```
(defthm wp-zcoef-g-instance
  (equal (wp-zcoef f1 x c low a f1save f2)
           (wp-zcoef-g f1 x c low a (* f1save f2) f2 8)))

(defthm floor-mod-rewrite
  (and (implies (and (equal d (* b c))
                             (rationalp a)
                             (rationalp b)
                             (rationalp c))
                      (equal (+ (* c (mod a b))
                                   (* d (floor a b)))
                              (* c a)))
         (implies (and (equal d (* b c))
                             (rationalp a)
                             (rationalp b)
                             (rationalp c)
                             (rationalp e))
                      (equal (+ (* c (mod a b))
                                  (* d (floor a b))
                                  e)
                              (+ (* c a) e)))))

(defthm mod-+-1
  (implies (and (equal (mod a 2) 0)
                      (natp i)
                      (integerp a))
               (equal (mod (+ 1 a) (expt 2 i))
                         (if (equal i 0)
                             0
                           (+ 1 (mod a (expt 2 i))))))
  :hints (("Subgoal 1"
           :use (:instance floor-floor-integer
                           (x (+ 1 a))
                           (i 2)
                           (j (expt 2 (+ -1 i))))
           :in-theory (e/d (mod)
                           (floor-floor-integer)))))

(defthm equal-transpose-constant
  (equal (equal (+ -1 a) 0)
           (equal a 1)))

(defthm wp-zcoef-g-multiplies
   (implies (and (not (zp x))
                       (integerp i)
                       (<= x i)
```

```
                    (natp f1)
                    (natp low)
                    (natp a)
                    (natp c)
                    (< low (expt 2 i))
                    (natp f2)
                    (< f2 (expt 2 i)))
                (equal (wp-zcoef-g f1 x c low a result f2 i)
                      (equal (+ (floor (+ low (* (expt 2 i) a) (* (expt 2 i) (expt 2 i) c))
                                      (expt 2 x))
                            (* f2
                              (mod f1 (expt 2 (1- x)))
                              (floor (expt 2 i) (expt 2 (1- x)))))
                            result)))
  :hints (("Goal"
         :in-theory (disable (:rewrite floor-zero . 3)
                             (:rewrite floor-zero . 4)
                             prefer-positive-addends-<
                             (:rewrite mod-x-y-=-x . 4)))))

(defthm wp-zcoef-is-correct
 (implies (and (natp f2)
               (< f2 256)
               (natp f1)
               (< f1 256)
               (natp low)
               (< low 256)
               (natp c))
          (wp-zcoef-1 f1 c low f2)))
```

;;; Proof Using the Loop Invariant Theory

```
(defthm wp-zcoef-g-instance
  (equal (wp-zcoef f1 x c low a f1save f2)
         (wp-zcoef-g f1 x c low a (* f1save f2) f2 8)))

;;; Package the state such that c, a and low are grouped into a long integer.

(defun low (a i) (mod (nfix a) (expt 2 (nfix i))))
(defun a (a i) (mod (floor (nfix a) (expt 2 (nfix i))) (expt 2 (nfix i))))
(defun c (a i) (floor (floor (nfix a) (expt 2 (nfix i))) (expt 2 (nfix i))))

(defun x (s) (nfix (car s)))
(defun f1 (s) (nfix (cadr s)))
(defun f2 (s) (nfix (caddr s)))
(defun result (s) (nfix (cadddr s)))
(defun i (s) (nfix (car (cddddr s))))
```

```
(defun lw (s) (nfix (cadddr (cddr s))))

(set-irrelevant-formals-ok t)

;;; Define an induction hint patterned after two copies of wp-zcoef-g.

(defun ind-2 (f1 x c low a f1p xp cp lowp ap f2 i)
  (declare (xargs :measure (dec x)))
  (if (equal (dec x) 0)
      0
    (ind-2 (+ (* (expt 2 (1- i)) (mod low 2))
              (floor f1 2))
           (dec x)
           (* (mod f1 2)
              (floor (+ (+ (* (expt 2 (1- i)) c) (floor a 2)) f2)
                     (expt 2 i)))
           (+ (* (expt 2 (1- i)) (mod a 2))
              (floor low 2))
           (if (equal (mod f1 2) 0)
              (+ (* (expt 2 (1- i)) c) (floor a 2))
            (mod (+ (+ (* (expt 2 (1- i)) c) (floor a 2)) f2)
                    (expt 2 i)))
            (+ (* (expt 2 (1- i)) (mod lowp 2))
               (floor f1p 2))
            (dec xp)
            (* (mod f1p 2)
               (floor (+ (+ (* (expt 2 (1- i)) cp) (floor ap 2)) f2)
                      (expt 2 i)))
            (+ (* (expt 2 (1- i)) (mod ap 2))
               (floor lowp 2))
            (if (equal (mod f1p 2) 0)
               (+ (* (expt 2 (1- i)) cp) (floor ap 2))
              (mod (+ (+ (* (expt 2 (1- i)) cp) (floor ap 2)) f2)
                      (expt 2 i)))
            f2
            i)))

(defthm equal-odd-even
  (implies (and (equal (mod a 2) (mod b 2))
                (integerp a)
                (integerp b))
           (not (equal (+ 1 a) b))))

(defthm mod-+-1
  (implies (and (equal (mod a 2) 0)
                (natp i)
                (integerp a))
               (equal (mod (+ 1 a) (expt 2 i))
                         (if (equal i 0)
                             0
                          (+ 1 (mod a (expt 2 i))))))
  :hints (("Goal"
           :in-theory (disable simplify-mod-+-mod))))

(defthm equal-wp-zcoef-g
  (implies (and (equal (mod f1 (expt 2 (1- x)))
                             (mod f1p (expt 2 (1- x))))
                    (equal x xp)
                    (equal (+ (* c (expt 2 i)) a)
                              (+ (* cp (expt 2 i)) ap))
                     (equal low lowp)
                     (<= x i)
                     (not (zp x))
                     (natp f1)
                     (natp x)
                     (natp c)
                     (natp low)
                     (natp a)
                     (natp f2)
                     (natp i)
                     (natp f1p)
                     (natp xp)
                     (natp cp)
                     (natp lowp)
                     (natp ap))
               (equal (equal (wp-zcoef-g f1 x c low a result f2 i)
                                   (wp-zcoef-g f1p xp cp lowp ap result f2 i))
                         t))
  :hints (("Goal"
           :induct (ind-2 f1 x c low a f1p xp cp lowp ap f2 i)
           :in-theory (disable (:rewrite mod-zero . 1)))))

;;; This compensates for an error within arithmetic-4

(defthm equal-transpose-constant
  (equal (equal (+ -1 a) 0)
            (equal a 1)))

(defthm expand-wp-zcoef-g
  (implies
   (< xp x)
   (and
    (equal (equal (wp-zcoef-g f1p xp cp lowp ap result f2 i)
                    (wp-zcoef-g f1 x c low a result f2 i))
            (equal (wp-zcoef-g f1p xp cp lowp ap result f2 i)
                   (if (equal (dec x) 0)
                       (equal (+ (* (+ (* (expt 2 (1- i)) c) (floor a 2)) (expt 2 i))
                                 (+ (* (expt 2 (1- i)) (mod a 2))
                                    (floor low 2)))
                              result)
                     (wp-zcoef-g (+ (* (expt 2 (1- i)) (mod low 2)) (floor f1 2))
                                 (dec x)
                                 (* (mod f1 2)
                                    (floor (+ (+ (* (expt 2 (1- i)) c) (floor a 2)) f2)
                                           (expt 2 i)))
                                 (+ (* (expt 2 (1- i)) (mod a 2)) (floor low 2))
                                 (if (equal (mod f1 2) 0)
                                    (+ (* (expt 2 (1- i)) c) (floor a 2))
                                  (mod (+ (+ (* (expt 2 (1- i)) c) (floor a 2)) f2)
                                       (expt 2 i)))
                                 result
                                 f2
                                 i))))
   (equal (equal (wp-zcoef-g f1 x c low a result f2 i)
                 (wp-zcoef-g f1p xp cp lowp ap result f2 i))
          (equal (if (equal (dec x) 0)
                       (equal (+ (* (+ (* (expt 2 (1- i)) c) (floor a 2)) (expt 2 i))
                                 (+ (* (expt 2 (1- i)) (mod a 2))
                                    (floor low 2)))
                              result)
                     (wp-zcoef-g (+ (* (expt 2 (1- i)) (mod low 2)) (floor f1 2))
                                 (dec x)
                                 (* (mod f1 2)
                                    (floor (+ (+ (* (expt 2 (1- i)) c) (floor a 2)) f2)
                                           (expt 2 i)))
                                 (+ (* (expt 2 (1- i)) (mod a 2)) (floor low 2))
                                 (if (equal (mod f1 2) 0)
                                    (+ (* (expt 2 (1- i)) c) (floor a 2))
                                  (mod (+ (+ (* (expt 2 (1- i)) c) (floor a 2)) f2)
                                       (expt 2 i)))
                                 result
                                 f2
                                 i))
                 (wp-zcoef-g f1p xp cp lowp ap result f2 i)))))
  :hints (("Goal"
         :expand (wp-zcoef-g f1 x c low a result f2 i)
         :hands-off (dec binary-+ binary-* expt mod floor))))

(defthm gather-minus-2
  (implies (integerp i)
           (and (equal (* -2 (expt 2 i)) (- (expt 2 (1+ i))))
                      (equal (* -2 a (expt 2 i)) (- (* a (expt 2 (1+ i)))))
                      (equal (* -2 (expt 2 i) a) (- (* (expt 2 (1+ i)) a)))
                      (equal (* -2 a (expt 2 i) b) (- (* a (expt 2 (1+ i)) b))))))

(defthm floor-floor-commutes
  (implies (and (rationalp a)
                (natp b)
                (natp c))
           (equal (floor (floor a b) c)
                  (floor (floor a c) b))))

(defthm floor-odometer-simple
  (implies (and (equal (mod b c) 0)
                (< a c)
                (natp b)
                (integerp c)
                (natp a)
                (integerp d))
           (equal (floor (+ a (* c d)) b)
                  (floor d (floor b c)))))

(defthm wp-zcoef-loop-invariant
 (implies (and (not (zp (x s)))
               (<= (x s) (i s))
               (< (f2 s) (expt 2 (i s)))
               (natp (car s))
               (natp (cadr s))
               (natp (caddr s))
               (natp (cadddr s))
               (natp (cadddr (cdr s)))
               (natp (cadddr (cddr s)))
               (equal (+ (floor (lw s) (expt 2 (x s)))
                         (* (f2 s)
                            (mod (f1 s) (expt 2 (1- (x s))))
                            (floor (expt 2 (i s)) (expt 2 (1- (x s))))))
                         (result s)))
          (wp-zcoef-g (f1 s)
                      (x s)
                      (c (lw s) (i s))
                      (low (lw s) (i s))
                      (a (lw s) (i s))
                      (result s)
                      (f2 s)
                      (i s)))
 :hints
 (("Goal"
   :use
```

```
((:functional-instance
  wp-is-weakest-invariant
  (b (lambda (s) (equal (dec (x s)) 0)))
  (qp (lambda (s) (equal (+ (* (+ (* (expt 2 (1- (i s))) (c (lw s) (i s)))
                                    (floor (a (lw s) (i s)) 2))
                                 (expt 2 (i s)))
                              (+ (* (expt 2 (1- (i s))) (mod (a (lw s) (i s)) 2))
                                 (floor (low (lw s) (i s)) 2)))
                           (result s))))
  (wp (lambda (s) (wp-zcoef-g (f1 s)
                              (x s)
                              (c (lw s) (i s))
                              (low (lw s) (i s))
                              (a (lw s) (i s))
                              (result s)
                              (f2 s)
                              (i s))))
  (measure (lambda (s) (dec (x s))))
  (sigma (lambda (s) (list (dec (x s))
                           (floor (f1 s) 2)
                           (f2 s)
                           (result s)
                           (i s)
                           (if (equal (mod (f1 s) 2) 0)
                               (floor (lw s) 2)
                             (+ (floor (lw s) 2) (* (f2 s) (expt 2 (i s))))))))
  (r (lambda (s) (and (not (zp (x s)))
                      (<= (x s) (i s))
                      (< (f2 s) (expt 2 (i s)))
                      (natp (car s))
                      (natp (cadr s))
                      (natp (caddr s))
                      (natp (cadddr s))
                      (natp (cadddr (cdr s)))
                      (natp (cadddr (cddr s)))
                      (equal (+ (floor (lw s) (expt 2 (x s)))
                                (* (f2 s)
                                   (mod (f1 s) (expt 2 (1- (x s))))
                                   (floor (expt 2 (i s)) (expt 2 (1- (x s))))))
                             (result s)))))))
 :in-theory (e/d (mod)
                 (wp-zcoef-g
                  equal-odd-even
                  floor-mod-elim)))

("Subgoal 2"
 :in-theory (disable wp-zcoef-g
                           equal-odd-even
                           floor-floor-integer
                           (:rewrite floor-zero . 3)
                           (:rewrite floor-zero . 4)
                           (:type-prescription nintegerp-expt)
                           (:type-prescription nintegerp-4b)
                           (:type-prescription mod-zero . 2)
                           (:rewrite mod-x-y-=-x . 4)))
  ("Subgoal 1"
   :hands-off (floor mod expt)
   :in-theory (enable wp-zcoef-g))))

(defthm wp-zcoef-loop-invariant-flat
 (implies (and (not (zp i))
                     (natp f1)
                     (natp low)
                     (natp a)
                     (natp c)
                     (< low (expt 2 i))
                     (< a (expt 2 i))
                     (natp f2)
                     (< f2 (expt 2 i))
                     (equal (+ (floor (+ low (* (expt 2 i) a) (* (expt 2 i) (expt 2 i) c))
                                     (expt 2 i))
                               (* f2
                                 (mod f1 (expt 2 (1- i)))
                                 (floor (expt 2 i) (expt 2 (1- i)))))
                           result))
              (wp-zcoef-g f1 i c low a result f2 i))
 :hints
 (("Goal"
   :in-theory (disable wp-zcoef-loop-invariant)
   :use (:instance
         wp-zcoef-loop-invariant
         (s (list i f1f2 result i (+ low (* (expt 2 i) a) (* (expt 2 i) (expt 2 i) c))))))))

(defthm wp-zcoef-is-correct
 (implies (and (natp f2)
                     (< f2 256)
                     (natp f1)
                     (< f1 256)
                     (natp low)
                     (< low 256)
                     (natp c))
             (wp-zcoef-1 f1 c low f2)))
```

;;; Proof Using the Tail Recursion Theory

```
(defthm wp-zcoef-g-instance
  (equal (wp-zcoef f1 x c low a f1save f2)
           (wp-zcoef-g f1 x c low a (* f1save f2) f2 8)))

;;; We partition the state into two halves, an "a" component that does the
;;; effective computation, and an "s" component that drives the computation.

;;; The "a" component is packaged into a long integer.

(defun low (a i) (mod a (expt 2 i)))
(defun a (a i) (mod (floor a (expt 2 i)) (expt 2 i)))
(defun c (a i) (floor a (expt 2 (* 2 i))))

;;; The "s" component is represented by a list.

(defun x (s) (car s))
(defun f1 (s) (cadr s))
(defun f2 (s) (caddr s))
(defun result (s) (cadddr s))
(defun i (s) (car (cddddr s)))

;;; Define the instantiation of h from the generic theory.  wp-zcoef-h multiplies
;;; in the standard way, except that it delivers twice the product.

(defun wp-zcoef-h (s)
  (declare (xargs :measure (dec (x s))))
  (if (equal (dec (x s)) 0)
      0
    (* 2 (+ (wp-zcoef-h (list (dec (x s))
                              (floor (f1 s) 2)
                              (f2 s)
                              (result s)
                              (i s)))
            (if (equal (mod (f1 s) 2) 0)
                0
              (nfix (f2 s)))))))

;;; Compute (h s), the bottom object under tau.

(defun btm-s (s)
  (declare (xargs :measure (dec (x s))))
  (if (equal (dec (x s)) 0)
      s
    (btm-s (list (dec (x s))
                 (floor (f1 s) 2)
                 (f2 s)
```

```
                   (result s)
                   (i s)))))

(set-irrelevant-formals-ok t)

;;; Define an induction hint patterned after two copies of wp-zcoef-g.

 (defun ind-2 (f1 x c low a f1p xp cp lowp ap f2 i)
  (declare (xargs :measure (dec x)))
  (if (equal (dec x) 0)
      0
    (ind-2 (+ (* (expt 2 (1- i)) (mod low 2))
                 (floor f1 2))
              (dec x)
              (* (mod f1 2)
                 (floor (+ (+ (* (expt 2 (1- i)) c) (floor a 2)) f2)
                        (expt 2 i)))
              (+ (* (expt 2 (1- i)) (mod a 2))
                 (floor low 2))
              (if (equal (mod f1 2) 0)
                 (+ (* (expt 2 (1- i)) c) (floor a 2))
               (mod (+ (+ (* (expt 2 (1- i)) c) (floor a 2)) f2)
                      (expt 2 i)))
               (+ (* (expt 2 (1- i)) (mod lowp 2))
                  (floor f1p 2))
               (dec xp)
               (* (mod f1p 2)
                  (floor (+ (+ (* (expt 2 (1- i)) cp) (floor ap 2)) f2)
                         (expt 2 i)))
               (+ (* (expt 2 (1- i)) (mod ap 2))
                  (floor lowp 2))
               (if (equal (mod f1p 2) 0)
                  (+ (* (expt 2 (1- i)) cp) (floor ap 2))
                 (mod (+ (+ (* (expt 2 (1- i)) cp) (floor ap 2)) f2)
                        (expt 2 i)))
               f2
               i)))

(defthm mod-expt-2
  (implies (and (natp a)
                    (natp b)
                    (not (zp i)))
              (equal (mod (+ a (* 2 b)) (expt 2 i))
                        (+ (mod a 2) (* 2 (mod (+ b (floor a 2)) (expt 2 (1- i)))))))
  :hints (("Goal"
           :in-theory (enable mod))
          ("Subgoal 1”"
          :use (:instance floor-floor-integer
                    (x (+ 1 (* 2 b) (* 2 k)))
                    (i 2)
                    (j (expt 2 (+ -1 i))))
          :in-theory (disable floor-floor-integer))))

(defthm equal-odd-even
  (implies (and (equal (mod a 2) (mod b 2))
                (integerp a)
                (integerp b))
           (not (equal (+ 1 a) b))))

(defthm equal-wp-zcoef-g
  (implies (and (equal (mod f1 (expt 2 (1- x)))
                       (mod f1p (expt 2 (1- x))))
                (equal x xp)
                (equal (+ (* c (expt 2 i)) a)
                       (+ (* cp (expt 2 i)) ap))
                 (equal low lowp)
                 (<= x i)
                 (not (zp x))
                 (natp f1)
                 (natp x)
                 (natp c)
                 (natp low)
                 (natp a)
                 (natp f2)
                 (natp i)
                 (natp f1p)
                 (natp xp)
                 (natp cp)
                 (natp lowp)
                 (natp ap))
           (equal (equal (wp-zcoef-g f1 x c low a result f2 i)
                         (wp-zcoef-g f1p xp cp lowp ap result f2 i))
                  t))
  :hints (("Goal"
          :induct (ind-2 f1 x c low a f1p xp cp lowp ap f2 i)
          :in-theory (disable (:rewrite mod-zero . 1)))))

;;; Necessary to correct a bug in arithmetic-4.

(defthm equal-transpose-constant
  (and (equal (equal (+ -1 a) 0)
              (equal a 1))
       (equal (equal (+ -2 a) 0)
              (equal a 2))))
```

```
(defthm floor-mod-rewrite
  (implies (and (natp a)
                (natp b)
                (equal c (floor a b)))
           (and (equal (+ (mod a b) (* b c)) a)
                (equal (+ (* b c) (mod a b)) a))))

(defthm to-mod
  (implies (and (integerp i)
                (< 0 i)
                (natp a))
           (equal (* (expt 2 (+ -1 i)) (floor a (expt 2 i)))
                  (- (floor a 2) (mod (floor a 2) (expt 2 (+ -1 i)))))))

(defthm mod-*-arg2-kb
  (implies (and (integerp i)
                (natp j)
                (integerp a)
                (equal b (floor a (expt 2 i))))
         (and (equal (+ (mod a (expt 2 i))
                        (* (expt 2 i) (mod b 2)))
                     (mod a (expt 2 (1+ i))))
              (equal (+ (mod a (expt 2 i))
                        (* (expt 2 i) (mod b (expt 2 j))))
                     (mod a (expt 2 (+ i j)))))))

(defthm floor-+-expt
  (implies (and (integerp a)
                (integerp b)
                (natp i)
                (natp j))
           (equal (floor (+ a (* b (expt 2 i)))
                         (expt 2 j))
                  (if (< i j)
                      (floor (+ b (floor a (expt 2 i)))
                             (expt 2 (- j i)))
                    (+ (floor a (expt 2 j))
                       (* b (expt 2 (- i j)))))))
  :hints (("Goal"
         :use ((:instance floor-floor-integer
                          (x (+ a (* b (expt 2 i))))
                          (i (expt 2 i))
                          (j (expt 2 (- j i))))
               (:instance floor-floor-integer
                          (x (+ a (* b (expt 2 i))))
                          (i (expt 2 j))
                      (j (expt 2 (- i j)))))
          :in-theory (disable floor-floor-integer))))

(defthm wp-zcoef-g=h
  (implies (and (natp (f1 s))
                (not (zp (x s)))
                (natp (i s))
                (<= (x s) (i s))
                (natp (f2 s))
                (< (f2 s) (expt 2 (i s)))
                (natp a))
         (equal (wp-zcoef-g (f1 s)
                            (x s)
                            (c a (i s))
                            (low a (i s))
                            (a a (i s))
                            (result s)
                            (f2 s)
                            (i s))
               (if (equal (dec (x s)) 0)
                   (equal (+ (* (+ (* (expt 2 (1- (i s))) (c a (i s)))
                                   (floor (a a (i s)) 2))
                                (expt 2 (i s)))
                             (+ (* (expt 2 (1- (i s))) (mod (a a (i s)) 2))
                                (floor (low a (i s)) 2)))
                          (result s))
                 (let ((a (floor (+ a (* (expt 2 (i s)) (wp-zcoef-h s)))
                               (expt 2 (1- (x s)))))
                      (s (btm-s s)))
                   (equal (+ (* (+ (* (expt 2 (1- (i s))) (c a (i s)))
                                   (floor (a a (i s)) 2))
                               (expt 2 (i s)))
                             (+ (* (expt 2 (1- (i s))) (mod (a a (i s)) 2))
                                (floor (low a (i s)) 2)))
                         (result s))))))
  :hints
  (("Goal"
    :use
    ((:functional-instance
      g=h
      (bb (lambda (s) (equal (dec (x s)) 0)))
      (qt (lambda (a s)
            (equal (+ (* (+ (* (expt 2 (1- (i s))) (c a (i s)))
                            (floor (a a (i s)) 2))
                         (expt 2 (i s)))
                      (+ (* (expt 2 (1- (i s))) (mod (a a (i s)) 2))
                         (floor (low a (i s)) 2)))
                (result s))))
   (g (lambda (a s) (wp-zcoef-g (f1 s)
                                (x s)
                                (c a (i s))
                                (low a (i s))
                                (a a (i s))
                                (result s)
                                (f2 s)
                                (i s))))
   (measure-g (lambda (s) (dec (x s))))
   (tau (lambda (s)
          (list (dec (x s))
                (floor (f1 s) 2)
                (f2 s)
                (result s)
                (i s))))
   (rho (lambda (a s)
          (if (equal (mod (f1 s) 2) 0)
              (floor a 2)
            (+ (floor a 2) (* (f2 s) (expt 2 (i s)))))))
   (rhoh (lambda (a s)
           (if (equal (mod (f1 s) 2) 0)
               (* 2 a)
             (* 2 (+ a (f2 s))))))
   (h (lambda (s) (wp-zcoef-h s)))
   (rt (lambda (a s)
         (and (natp (f1 s))
              (not (zp (x s)))
              (natp (i s))
              (<= (x s) (i s))
              (natp (f2 s))
              (< (f2 s) (expt 2 (i s)))
              (natp a))))
   (id (lambda () 0))
   (op (lambda (a x s)
         (if (equal (dec (x s)) 0)
             a
           (floor (+ a (* (expt 2 (i s)) x)) (expt 2 (1- (x s)))))))
   (hs (lambda (s) (btm-s s))))))
 ("Subgoal 2"
  :expand (wp-zcoef-g (cadr s)
                      (car s)
                      (floor a (expt 2 (* 2 (cadddr (cdr s)))))
                      (mod a (expt 2 (cadddr (cdr s))))
                      (mod (floor a (expt 2 (cadddr (cdr s))))
                      (expt 2 (cadddr (cdr s))))
                      (cadddr s)
```

```
                                (caddr s)
                                (cadddr (cdr s)))
  :in-theory (disable wp-zcoef-g))))

(defthm btm-s-destruct
  (implies (and (not (zp (x s)))
                (natp (cadr s))
                (natp (caddr s))
                (natp (cadddr s))
                (natp (cadddr (cdr s))))
           (and (equal (car (btm-s s)) 1)
                (equal (cadr (btm-s s))
                       (floor (f1 s) (expt 2 (1- (x s)))))
                (equal (caddr (btm-s s)) (f2 s))
                (equal (cadddr (btm-s s)) (result s))
                (equal (car (cddddr (btm-s s))) (i s)))))

(defthm floor-mod-*-2-kb
  (implies (and (equal c (floor a 2))
                (equal b (* 2 d))
                (rationalp a)
                (rationalp d))
           (equal (+ (* b c)
                     (* d (mod a 2)))
                  (* d a))))

(defthm wp-zcoef-g=h-flat
  (implies (and (not (zp i))
                (natp f1)
                (natp result)
                (natp f2)
                (< f2 (expt 2 i))
                (natp a)
                (< a (expt 2 i))
                (natp low)
                (< low (expt 2 i)))
           (equal (wp-zcoef-g f1 i 0 low a result f2 i)
                  (if (equal (dec i) 0)
                      (equal (+ (* (floor a 2) (expt 2 i))
                                (* (mod a 2) (expt 2 (1- i)))
                                (floor low 2))
                             result)
                    (equal (+ a (wp-zcoef-h (list i f1 f2 result i)))
                           result))))
  :hints (("Goal"
           :use (:instance wp-zcoef-g=h
                  (a (+ low (* (expt 2 i) a)))
```

```
                    (s (list i f1 f2 result i)))
         :in-theory (disable wp-zcoef-g=h
                              btm-s
                              wp-zcoef-g
                              wp-zcoef-h))))

(defthm equal-transpose-expt
  (implies (and (rationalp a)
                (rationalp b)
                (rationalp c))
           (equal (equal a (* b c (expt 1/2 i)))
                  (equal (* a (expt 2 i)) (* b c))))
  :hints (("Goal"
         :use (:instance (:theorem
                          (implies (and (rationalp a)
                                        (rationalp b)
                                        (rationalp c)
                                        (not (equal c 0)))
                                   (equal (equal a b)
                                          (equal (* c a) (* c b)))))
                    (a a)
                    (b (* b c (expt 1/2 i)))
                    (c (expt 2 i))))))

(defthm wp-zcoef-h-multiplies
  (implies (and (not (zp (x s)))
                (<= (x s) (i s))
                (natp (car s))
                (natp (cadr s))
                (natp (caddr s))
                (natp (cadddr s))
                (natp (cadddr (cdr s))))
           (equal (wp-zcoef-h s)
                  (* 2 (f2 s) (mod (f1 s) (expt 2 (1- (x s)))))))
  :hints (("Goal"
         :in-theory (enable mod))))

(defthm wp-zcoef-is-correct
  (implies (and (natp f2)
                (< f2 256)
                (natp f1)
                (< f1 256)
                (natp low)
                (< low 256)
                (natp c))
           (wp-zcoef-1 f1 c low f2)))
```

;;; Proof Using the Alternative Induction Theory

```
(defthm wp-zcoef-g-instance
  (equal (wp-zcoef f1 x c low a f1save f2)
           (wp-zcoef-g f1 x c low a (* f1save f2) f2 8)))

;;; Inspection of the assembly program shows that c effectively behaves as
;;; an extension of a.  Combining c and a into
;;;
;;;      ac = a + 2^i*c
;;;
;;; both simplifies the body of wp-zcoef-ac and enables easy expression of an
;;; alternate induction.

(defun wp-zcoef-ac (f1 x ac low result f2 i)
  (declare (xargs :measure (dec x)))
  (if (equal (dec x) 0)
      (equal (+ (floor low 2) (* ac (expt 2 (1- i))))
             result)
    (wp-zcoef-ac (+ (floor f1 2) (* (expt 2 (1- i)) (mod low 2)))
                 (dec x)
                 (if (equal (mod f1 2) 0)
                     (floor ac 2)
                   (+ f2 (floor ac 2)))
                 (+ (floor low 2) (* (expt 2 (1- i)) (mod ac 2)))
                 result
                 f2
                 i)))

;;; Use the alternative induction generic theory to prove an equivalence between
;;; wp-zcoef-g and wp-zcoef-ac.  Represent the state s as a list of state variables,
;;; and provide accessor functions.  Notice that the state includes all variables in
;;; wp-zcoef-g and wp-zcoef-ac.

(defun f1 (s) (car s))
(defun x (s) (cadr s))
(defun c (s) (caddr s))
(defun low (s) (cadddr s))
(defun a (s) (car (cddddr s)))
(defun result (s) (cadr (cddddr s)))
(defun f2 (s) (caddr (cddddr s)))
(defun i (s) (cadddr (cddddr s)))
(defun ac (s) (car (cddddr (cddddr s))))

;;; Prove an instance of fn1-as-fn2.

(defthm wp-zcoef-g-as-ac
(let ((ts (list (f1 s)
                (x s)
                (c s)
                (low s)
                (a s)
                (result s)
                (f2 s)
                (i s)
                (+ (a s) (* (c s) (expt 2 (i s)))))))
  (implies (and (not (zp (x s)))
                (integerp (i s))
                (natp (f1 s))
                (natp (c s))
                (natp (low s))
                (natp (a s))
                (natp (result s))
                (natp (f2 s))
                (natp (ac s))
                (not (< (i s) (x s))))
           (equal (wp-zcoef-g (f1 s) (x s) (c s) (low s) (a s) (result s) (f2 s) (i s))
                  (wp-zcoef-ac (f1 ts) (x ts) (ac ts) (low ts) (result ts) (f2 ts) (i ts)))))
:hints
(("Goal"
  :use
  (:functional-instance
   fn1-as-fn2
   (b1 (lambda (s) (equal (dec (x s)) 0)))
   (b2 (lambda (s) (equal (dec (x s)) 0)))
   (q1 (lambda (s) (equal (+ (* (+ (* (expt 2 (1- (i s))) (c s))
                                   (floor (a s) 2))
                                (expt 2 (i s)))
                             (+ (* (expt 2 (1- (i s))) (mod (a s) 2))
                                (floor (low s) 2)))
                          (result s))))
   (q2 (lambda (s) (equal (+ (floor (low s) 2)
                             (* (ac s) (expt 2 (1- (i s)))))
                          (result s))))
   (p (lambda (s) (and (not (zp (x s)))
                       (integerp (i s))
                       (natp (f1 s))
                       (natp (c s))
                       (natp (low s))
                       (natp (a s))
                       (natp (result s))
                       (natp (f2 s))
                       (natp (ac s))
                       (not (< (i s) (x s))))))
```

```
(fn1 (lambda (s)
       (wp-zcoef-g (f1 s) (x s) (c s) (low s) (a s) (result s) (f2 s) (i s))))
(fn2 (lambda (s)
       (wp-zcoef-ac (f1 s) (x s) (ac s) (low s) (result s) (f2 s) (i s))))
(sigma1 (lambda (s)
          (list
           (+ (* (expt 2 (1- (i s))) (mod (low s) 2))
              (floor (f1 s) 2))
           (dec (x s))
           (if (equal (mod (f1 s) 2) 0)
               0
             (floor
              (+ (+ (* (expt 2 (1- (i s))) (c s))
                    (floor (a s) 2))
                 (f2 s))
              (expt 2 (i s))))
           (+ (* (expt 2 (1- (i s))) (mod (a s) 2))
              (floor (low s) 2))
           (if (equal (mod (f1 s) 2) 0)
               (+ (* (expt 2 (1- (i s))) (c s))
                  (floor (a s) 2))
             (mod
              (+ (+ (* (expt 2 (1- (i s))) (c s))
                    (floor (a s) 2))
                 (f2 s))
              (expt 2 (i s))))
           (result s)
           (f2 s)
           (i s)
           (ac s))))
(sigma2 (lambda (s)
          (list
           (+ (* (expt 2 (1- (i s))) (mod (low s) 2))
              (floor (f1 s) 2))
           (dec (x s))
           (if (equal (mod (f1 s) 2) 0)
               0
             (floor
              (+ (+ (* (expt 2 (1- (i s))) (c s))
                    (floor (a s) 2))
                 (f2 s))
              (expt 2 (i s))))
           (+ (* (expt 2 (1- (i s))) (mod (ac s) 2))
              (floor (low s) 2))
           (if (equal (mod (f1 s) 2) 0)
               (+ (* (expt 2 (1- (i s))) (c s))
                  (floor (a s) 2))
                  (mod
                   (+ (+ (* (expt 2 (1- (i s))) (c s))
                         (floor (a s) 2))
                      (f2 s))
                   (expt 2 (i s))))
                (result s)
                (f2 s)
                (i s)
                (if (equal (mod (f1 s) 2) 0)
                    (floor (ac s) 2)
                  (+ (f2 s) (floor (ac s) 2))))))
     (measure1 (lambda (s) (if (zp (x s)) 256 (x s))))
     (id-alt
      (lambda (s)
        (list
         (f1 s)
         (x s)
         (c s)
         (low s)
         (a s)
         (result s)
         (f2 s)
         (i s)
         (+ (a s) (* (c s) (expt 2 (i s))))))))))

;;; Convert the above instantiation into an effective rewrite rule.

(defthm wp-zcoef-g-as-ac-rewrite
    (implies (and (not (zp x))
                        (integerp i)
                        (natp f1)
                        (natp c)
                        (natp low)
                        (natp a)
                        (natp result)
                        (natp f2)
                        (not (< i x)))
                 (equal (wp-zcoef-g f1 x c low a result f2 i)
                           (wp-zcoef-ac f1 x (+ a (* c (expt 2 i))) low result f2 i)))
    :hints
    (("Goal"
      :use (:instance wp-zcoef-g-as-ac
                     (s (list f1 x c low a result f2 i (+ a (* c (expt 2 i))))))
      :hands-off (floor mod expt))))

(defthm equal-odd-even
  (implies (and (equal (mod a 2) (mod b 2))
                    (integerp a)
                    (integerp b))
               (not (equal (+ 1 a) b))))

(defthm mod-+-1
  (implies (and (equal (mod a 2) 0)
                    (natp i)
                    (integerp a))
               (equal (mod (+ 1 a) (expt 2 i))
                         (if (equal i 0)
                              0
                           (+ 1 (mod a (expt 2 i))))))
  :hints (("Goal"
           :in-theory (disable simplify-mod-+-mod))))

;;; We now look for a substitution id-alt, which leaves wp-zcoef-ac invariant.
;;; Since we will be proving that sigma1 and id-alt commute, we would indeed have
;;; a simple proof if sigma1 and id-alt altered disjoint sets of variables.  Only
;;; f2, result and i are left unchanged by sigma1.  Looking at the assembly
;;; language program, we see that a change in f2 only affects ac.  So if we
;;; decremented f2 and incremented ac whenever (f1 mod 2) = 1, the computation
;;; would be unchanged.  Notice that a single change in f2 could affect the
;;; computation on the next x - 1 iterations.  So it is necessary to add
;;; (* 2 (mod f1 (expt 2 (1- x)))) to ac.  This defines id-alt.

(defthm f2-induction
  (let ((ts
        (list (f1 s)
              (x s)
              (c s)
              (low s)
              (a s)
              (result s)
              (1- (f2 s))
              (i s)
              (+ (ac s) (* 2 (mod (f1 s) (expt 2 (1- (x s)))))))))
    (implies
     (and (not (zp (f2 s)))
            (not (zp (x s)))
            (integerp (i s))
            (natp (f1 s))
            (natp (c s))
            (natp (low s))
            (natp (a s))
            (natp (result s))
            (natp (ac s))
            (not (< (i s) (x s))))
    (equal (wp-zcoef-ac (f1 s) (x s) (ac s) (low s) (result s) (f2 s) (i s))
           (wp-zcoef-ac (f1 ts) (x ts) (ac ts) (low ts) (result ts) (f2 ts) (i ts)))))
  :hints
  (("Goal"
    :use
    (:functional-instance
     fn1-as-fn2
     (b1 (lambda (s) (equal (dec (x s)) 0)))
     (b2 (lambda (s) (equal (dec (x s)) 0)))
     (q1 (lambda (s) (equal (+ (floor (low s) 2) (* (ac s) (expt 2 (1- (i s)))))
                            (result s))))
     (q2 (lambda (s) (equal (+ (floor (low s) 2) (* (ac s) (expt 2 (1- (i s)))))
                            (result s))))
     (p (lambda (s) (and (not (zp (f2 s)))
                         (not (zp (x s)))
                         (integerp (i s))
                         (natp (f1 s))
                         (natp (c s))
                         (natp (low s))
                         (natp (a s))
                         (natp (result s))
                         (natp (ac s))
                         (not (< (i s) (x s))))))
     (fn1 (lambda (s)
          (wp-zcoef-ac (f1 s) (x s) (ac s) (low s) (result s) (f2 s) (i s))))
     (fn2 (lambda (s)
          (wp-zcoef-ac (f1 s) (x s) (ac s) (low s) (result s) (f2 s) (i s))))
     (sigma1 (lambda (s)
              (list
               (+ (* (expt 2 (1- (i s))) (mod (low s) 2)) (floor (f1 s) 2))
               (dec (x s))
               (c s)
               (+ (* (expt 2 (1- (i s))) (mod (ac s) 2)) (floor (low s) 2))
               (a s)
               (result s)
               (f2 s)
               (i s)
               (if (equal (mod (f1 s) 2) 0)
                   (floor (ac s) 2)
                 (+ (f2 s) (floor (ac s) 2))))))
     (sigma2 (lambda (s)
              (list
               (+ (* (expt 2 (1- (i s))) (mod (low s) 2)) (floor (f1 s) 2))
               (dec (x s))
               (c s)
               (+ (* (expt 2 (1- (i s))) (mod (ac s) 2)) (floor (low s) 2))
               (a s)
```

```
               (result s)
               (f2 s)
               (i s)
               (if (equal (mod (f1 s) 2) 0)
                   (floor (ac s) 2)
                 (+ (f2 s) (floor (ac s) 2))))))
     (measure1 (lambda (s) (if (zp (x s)) 256 (x s))))
     (id-alt
      (lambda (s)
        (list
         (f1 s)
         (x s)
         (c s)
         (low s)
         (a s)
         (result s)
         (1- (f2 s))
         (i s)
         (+ (ac s) (* 2 (mod (f1 s) (expt 2 (1- (x s))))))))))))

;;; Convert the above to an effective rewrite rule.

(defthm f2-induction-rewrite
   (implies
    (and (not (zp f2))
           (not (zp x))
           (natp f1)
           (natp c)
           (natp a)
           (natp ac)
           (natp low)
           (natp result)
           (integerp i)
           (not (< i x)))
    (equal (wp-zcoef-ac f1 x ac low result f2 i)
             (wp-zcoef-ac f1
                            x
                            (+ ac (* 2 (mod f1 (expt 2 (1- x)))))
                            low
                            result
                            (1- f2)
                            i)))
   :hints
   (("Goal"
     :use
     (:instance f2-induction
               (s (list f1 x c low a result f2 i ac))))))
```

```
;;; This compensates for an error within arithmetic-4

(defthm equal-transpose-constant
  (equal (equal (+ -1 a) 0)
            (equal a 1)))

;;; This is the base case for the alternative induction.  Its form is readily apparent,
;;; since wp-zcoef-ac simply right shifts ac and low x times when f2 = 0.

(defthm f2-induction-base-case
  (implies (and (equal f2 0)
                      (not (zp x))
                      (natp f1)
                      (natp ac)
                      (natp low)
                      (natp result)
                      (integerp i)
                      (not (< i x)))
              (equal (wp-zcoef-ac f1 x ac low result f2 i)
                        (equal (+ (floor low (expt 2 x)) (* ac (expt 2 (- i x))))
                                  result)))
  :hints
  (("Goal"
    :induct (wp-zcoef-ac f1 x ac low result f2 i)
    :do-not '(generalize))))

(set-irrelevant-formals-ok t)

;;; This is the induction hint corresponding to id-alt

(defun wp-ind-id-alt (f1 x ac f2)
  (if (zp f2)
      t
    (wp-ind-id-alt f1 x (+ ac (* 2 (mod f1 (expt 2 (1- x))))) (1- f2))))

;;; Rewrite an arbitrary application of wp-zcoef-ac into the base case.

(defthm wp-zcoef-ac-as-0
  (implies
   (and (not (zp x))
           (natp f1)
           (natp ac)
           (natp low)
           (natp result)
           (natp f2)
           (integerp i)
           (not (< i x)))
   (equal (wp-zcoef-ac f1 x ac low result f2 i)
             (if (zp f2)
                 (equal (+ (floor low (expt 2 x)) (* ac (expt 2 (- i x))))
                          result)
            (wp-zcoef-ac
             f1
             x
             (+ ac (* 2 f2 (mod f1 (expt 2 (1- x)))))
             low
             result
             0
             i))))
  :hints
  (("Goal"
    :induct (wp-ind-id-alt f1 x ac f2))))

;;; Finally, the correctness result.

(defthm mult-program-is-correct
  (implies (and (< low 256)
                (< f1 256)
                (< f2 256)
                (natp f1)
                (natp c)
                (natp low)
                (natp f2))
           (wp-zcoef-1 f1 c low f2)))
```

Appendix F

The Multiply Program - NQTHM

```
;;; The following is a Floyd-Hoare correctness specification for the multiply
;;; program.
;;;
;;;     { F1=F1SAVE ^ F1<256 ^ F2<256 ^ LOW<256 }
;;;
;;;             LDX #8          load the X register immediate with the value 8
;;;             LDA #0          load the A register immediate with the value 0
;;; LOOP     ROR F1          rotate F1 right circular through the carry flag
;;;             BCC ZCOEF  branch on carry flag clear to ZCOEF
;;;             CLC             clear the carry flag
;;;             ADC F2          add with carry F2 to the contents of A
;;; ZCOEF   ROR A           rotate A right circular through the carry flag
;;;             ROR LOW     rotate LOW right circular through the carry flag
;;;             DEX             decrement the X register by 1
;;;             BNE LOOP    branch if X is non-zero to LOOP
;;;
;;;     { LOW + 256*A = F1SAVE*F2 }

;;; Provide semantics for the Mostek 6502 DEX instruction.  The remaining
;;; instructions have semantics built into the weakest precondition generation
;;; program.

(defn dec (x)
  (if (not (zerop x))
     (sub1 x)
   255))

;;; This is mechanically derived.

(DEFN WP-ZCOEF (F1 X C LOW A F1SAVE F2)
      (IF (EQUAL (DEC X) 0)
           (EQUAL
            (PLUS (TIMES (PLUS (TIMES 128 C) (QUOTIENT A 2)) 256)
                     (PLUS (TIMES 128 (REMAINDER A 2))
                              (QUOTIENT LOW 2)))
            (TIMES F1SAVE F2))
          (WP-ZCOEF
            (PLUS (TIMES 128 (REMAINDER LOW 2)) (QUOTIENT F1 2))
            (DEC X)
            (TIMES (REMAINDER F1 2)
                      (QUOTIENT (PLUS (PLUS (TIMES 128 C) (QUOTIENT A 2)) F2)
                                    256))
            (PLUS (TIMES 128 (REMAINDER A 2))
                       (QUOTIENT LOW 2))
              (IF (EQUAL (REMAINDER F1 2) 0)
                    (PLUS (TIMES 128 C) (QUOTIENT A 2))
                  (REMAINDER (PLUS (PLUS (TIMES 128 C) (QUOTIENT A 2)) F2)
                                        256))
              F1SAVE
              F2))
       ((lessp (dec x)))) ; This hint is user added

;;; Weakest precondition at the beginning of the program

(defn wp-zcoef-1 (f1 c low f2)
  (wp-zcoef
   (plus (times 128 c) (quotient f1 2))
   8
   0
   low
   (times (remainder f1 2) f2)
   f1
   f2))

;;; (exp i b) = b**i if b > 0, otherwise 0.

(defn exp (i b)
  (if (zerop b)
      0
    (if (zerop i)
        1
      (times b (exp (sub1 i) b)))))

;;; Generalize the register size in order to capture properties of 128 and 256.

(defn wp-zcoef-g (f1 x c low a result f2 i)
       (if (equal (dec x) 0)
           (equal (plus (times (plus (times (exp (sub1 i) 2) c) (quotient a 2))
                                (exp i 2))
                          (plus (times (exp (sub1 i) 2) (remainder a 2))
                                 (quotient low 2)))
                     result)
          (wp-zcoef-g
             (plus (times (exp (sub1 i) 2) (remainder low 2)) (quotient f1 2))
             (dec x)
             (times (remainder f1 2)
                      (quotient (plus (plus (times (exp (sub1 i) 2) c) (quotient a 2)) f2)
                                   (exp i 2)))
             (plus (times (exp (sub1 i) 2) (remainder a 2)) (quotient low 2))
             (if (equal (remainder f1 2) 0)
                  (plus (times (exp (sub1 i) 2) c) (quotient a 2))
                 (remainder (plus (plus (times (exp (sub1 i) 2) c) (quotient a 2)) f2)
                            (exp i 2)))
             result
             f2
             i))
       ((lessp (dec x))))
```

;;; Proof by Generalization

```
;;; Transform wp-zcoef-1 into an instance of the more general function.

(prove-lemma wp-zcoef-g-instance (rewrite)
  (equal (wp-zcoef f1 x c low a f1save f2)
         (wp-zcoef-g f1 x c low a (times f1save f2) f2 8))
   ((hands-off plus times quotient remainder difference)
    (expand (wp-zcoef f1 0 c low a f1save f2)
            (wp-zcoef-g f1 0 c low a (times f1save f2) f2 8))))

;;; An alternative to remainder-plus-arg1 that generates fewer case splits.

(prove-lemma remainder-plus-arg1-alt (rewrite)
   (implies (equal (remainder a d) 0)
            (and (equal (remainder (plus a b) d) (remainder b d))
                 (equal (remainder (plus b a) d) (remainder b d))
                 (equal (remainder (plus b a c) d) (remainder (plus b c) d))
                 (equal (remainder (plus b c a) d) (remainder (plus b c) d))))
    ((disable-theory if-normalization)
     (hands-off difference)
     (disable times-add1)))

;;; An alternative to quotient-plus-arg1 that generates fewer case splits.

(prove-lemma quotient-plus-arg1-alt (rewrite)
   (implies (equal (remainder a d) 0)
            (and (equal (quotient (plus a b) d) (plus (quotient a d) (quotient b d)))
                 (equal (quotient (plus b a) d) (plus (quotient a d) (quotient b d)))
                 (equal (quotient (plus b a c) d) (plus (quotient a d) (quotient (plus b c) d)))
                (equal (quotient (plus b c a) d) (plus (quotient a d) (quotient (plus b c) d)))))
    ((disable-theory if-normalization)
     (hands-off difference)
     (disable lessp-transpose-meta
              equal-transpose-meta
              remainder-difference-arg1
              remainder-plus-arg1)))

;;; In order to preserve the identiy of the constant 2, we must disable times-add1 and times.
;;; The following meta rule, selectively applies times-add1 to non constant arguments.

(defn times-add1-fcn (a)
  (if (equal (car a) 'times)
      (if (equal (caadr a) 'add1)
          `(plus ,(caddr a) (times ,(cadadr a) ,(caddr a)))
        (if (equal (caaddr a) 'add1)
            `(plus ,(cadr a) (times ,(cadr a) ,(cadaddr a)))
          a))
    a))

(prove-lemma times-add1-meta ((meta times))
             (equal (eval$ t a y)
                    (eval$ t (times-add1-fcn a) y)))

(prove-lemma remainder-exp-exp (rewrite)
  (implies (not (lessp i j))
               (equal (remainder (exp i 2) (exp j 2)) 0))
  ((disable times
                times-add1)))

(prove-lemma quotient-exp-2 (rewrite)
   (equal (quotient (exp i 2) 2)
             (if (zerop i)
                  0
                (exp (sub1 i) 2))))

;;; Describe effect of wp-zcoef-g on arbitrary c, a and low.

(prove-lemma wp-zcoef-g-multiplies (rewrite)
   (implies (and (not (zerop x))
                      (not (lessp i x))
                      (lessp low (exp i 2))
                      (lessp f2 (exp i 2)))
                (equal (wp-zcoef-g f1 x c low a result f2 i)
                          (equal
                             (plus (quotient (plus low (times a (exp i 2)) (times c (exp i 2) (exp i 2)))
                                                   (exp x 2))
                                     (times f2
                                              (remainder f1 (exp (sub1 x) 2))
                                              (quotient (exp i 2) (exp (sub1 x) 2))))
                             result)))
   ((disable-theory if-normalization)
    (hands-off difference)
    (induct (wp-zcoef-g f1 x c low a result f2 i))
    (disable remainder-add1-arg2
                quotient-add1-arg2
                quotient-plus-arg1
                 remainder-plus-arg1
                 remainder-plus-arg1-simple
                 times
                 times-add1
                 equal-add1
                 lessp-2
                 sub1-times
                 sub1-remainder
                 sub1-quotient
                 remainder-difference-arg1
                 remainder-minus-one-as-0
                 lessp-times-single-linear
                 lessp-odometer-simple
                 equal-odometer-simple
                 lessp-quotient
                 no-divisors-of-zero
                 equal-quotient-0
                 plus-is-0
                 sub1-plus)))

(prove-lemma wp-zcoef-is-correct (rewrite)
   (implies (and (lessp f2 256)
                      (lessp f1 256)
                      (lessp low 256))
                (wp-zcoef-1 f1 c low f2))
  ((disable-theory if-normalization)
    (hands-off difference)
    (disable times
                 times-add1
                 quotient-add1-arg2
                 remainder-add1-arg2)))
```

;;; Proof Using the Loop Invariant Theory

```
;;; Define state accessors.

(defn x (s) (car s))                               ; x register counter
(defn f1 (s) (cadr s))                             ; first factor
(defn f2 (s) (caddr s))                            ; second factor
(defn result (s) (cadddr s))                       ; the result
(defn i (s) (caddddr s))                           ; word size in bits
(defn w (s) (cadddddr s))                          ; long word accumulator

;;; Define components of w.

(defn c (a i) (quotient (quotient a (exp i 2)) (exp i 2)))     ; the carry flag
```

```
(defn a (a i) (remainder (quotient a (exp i 2)) (exp i 2)))   ; the accumulator
(defn low (a i) (remainder a (exp i 2)))                       ; lower half of product

;;; Define an induction hint which recurses on two copies of wp-zcoef-g.

(defn ind-2 (f1 x c low a f1p xp cp lowp ap f2 i)
  (if (equal (dec x) 0)
      0
    (ind-2
     (plus (times (exp (sub1 i) 2) (remainder low 2)) (quotient f1 2))
     (dec x)
     (times (remainder f1 2)
            (quotient (plus (plus (times (exp (sub1 i) 2) c) (quotient a 2)) f2)
                      (exp i 2)))
     (plus (times (exp (sub1 i) 2) (remainder a 2)) (quotient low 2))
     (if (equal (remainder f1 2) 0)
         (plus (times (exp (sub1 i) 2) c) (quotient a 2))
             (remainder (plus (plus (times (exp (sub1 i) 2) c) (quotient a 2)) f2)
                        (exp i 2)))
     (plus (times (exp (sub1 i) 2) (remainder lowp 2)) (quotient f1p 2))
     (dec xp)
     (times (remainder f1p 2)
            (quotient (plus (plus (times (exp (sub1 i) 2) cp) (quotient ap 2)) f2)
                      (exp i 2)))
     (plus (times (exp (sub1 i) 2) (remainder ap 2)) (quotient lowp 2))
     (if (equal (remainder f1p 2) 0)
         (plus (times (exp (sub1 i) 2) cp) (quotient ap 2))
             (remainder (plus (plus (times (exp (sub1 i) 2) cp) (quotient ap 2)) f2)
                        (exp i 2)))
     f2
     i))
  ((lessp (dec x))))

;;; An alternative to remainder-plus-arg1 that generates fewer case splits.

(prove-lemma remainder-plus-arg1-alt (rewrite)
   (implies (equal (remainder a d) 0)
            (and (equal (remainder (plus a b) d) (remainder b d))
                 (equal (remainder (plus b a) d) (remainder b d))
                 (equal (remainder (plus b a c) d) (remainder (plus b c) d))
                 (equal (remainder (plus b c a) d) (remainder (plus b c) d))))
   ((disable-theory if-normalization)
    (hands-off difference)
    (disable times-add1)))

;;; A simple fact absent from modularithmetic-98.
```

```
(prove-lemma equal-even-odd (rewrite)
   (implies (equal (remainder a 2) (remainder b 2))
                (and (not (equal a (add1 b)))
                        (not (equal (add1 b) a))))
   ((disable-theory if-normalization)
    (disable remainder-add1-arg2)))

;;; Backchain on the arguments when proving an equality between function applications.

(prove-lemma equal-wp-zcoef-g (rewrite)
   (implies (and (equal (remainder f1 (exp (sub1 x) 2)) (remainder f1p (exp (sub1 x) 2)))
                      (equal x xp)
                      (equal (plus (times c (exp i 2)) a) (plus (times cp (exp i 2)) ap))
                      (equal low lowp)
                      (not (lessp i x))
                      (not (zerop x)))
                 (equal (equal (wp-zcoef-g f1 x c low a result f2 i)
                                    (wp-zcoef-g f1p xp cp lowp ap result f2 i))
                            t))
   ((disable-theory if-normalization)
    (hands-off difference)
    (induct (ind-2 f1 x c low a f1p xp cp lowp ap f2 i))
    (expand (exp (sub1 xp) 2))
    (disable remainder-add1-arg2
                quotient-add1-arg2
                quotient-plus-arg1
                remainder-plus-arg1
                remainder-plus-arg1-simple
                times
                times-add1
                sub1-times
                sub1-plus
                sub1-remainder
                sub1-quotient
                remainder-times-arg1
                remainder-difference-arg1
                equal-add1
                remainder-minus-one-as-0
                equal-transpose-meta
                quotient-is-unique
                lessp-quotient-arg2-linear
                lessp-quotient
                plus-is-0
                remainder-add1-arg1
                equal-quotient-0
                quotient-times-arg1-simple
                quotient-add1-arg1)))
```

```
(prove-lemma remainder-exp-exp (rewrite)
  (implies (not (lessp i j))
              (equal (remainder (exp i 2) (exp j 2)) 0))
  ((disable times
              times-add1)))

;;; An alternative to quotient-plus-arg1 that generates fewer case splits.

(prove-lemma quotient-plus-arg1-alt (rewrite)
   (implies (equal (remainder a d) 0)
              (and (equal (quotient (plus a b) d) (plus (quotient a d) (quotient b d)))
                     (equal (quotient (plus b a) d) (plus (quotient a d) (quotient b d)))
                     (equal (quotient (plus b a c) d) (plus (quotient a d) (quotient (plus b c) d)))
                    (equal (quotient (plus b c a) d) (plus (quotient a d) (quotient (plus b c) d)))))
   ((disable-theory if-normalization)
    (hands-off difference)
    (disable lessp-transpose-meta
              equal-transpose-meta
              equal-even-odd
              remainder-difference-arg1
              remainder-plus-arg1)))

;;; This should actually be part of modularithmetic-98.

(prove-lemma remainder-times-arg2-kb (rewrite)
   (equal (plus (remainder a b) (times b (remainder (quotient a b) c)))
            (plus (remainder a c) (times c (remainder (quotient a c) b))))
   ((use (remainder-times-arg2
                (a a)
                (zb b)
                (zc c))
            (remainder-times-arg2
                (a a)
                (zb c)
                (zc b)))
    (disable remainder-times-arg2)
    (hands-off quotient remainder)))

(prove-lemma quotient-remainder-kb (rewrite)
   (equal (plus (remainder (quotient a b) c) (times c (quotient (quotient a c) b)))
            (quotient a b)))

(functionally-instantiate wp-zcoef-loop-invariant nil
  (implies (and (not (zerop (x s)))
                   (not (lessp (i s) (x s)))
                   (lessp (f2 s) (exp (i s) 2))
```

```
                    (equal (plus (quotient (w s) (exp (x s) 2))
                                 (times (f2 s) (remainder (f1 s) (exp (sub1 (x s)) 2))
                                 (quotient (exp (i s) 2) (exp (sub1 (x s)) 2))))
                           (result s)))
                 (wp-zcoef-g
                  (f1 s)
                  (x s)
                  (c (w s) (i s))
                  (low (w s) (i s))
                  (a (w s) (i s))
                  (result s)
                  (f2 s)
                  (i s)))
wp-is-weakest-invariant
((b (lambda (s) (equal (dec (x s)) 0)))
 (q (lambda (s) (equal (plus (times (plus (times (exp (sub1 (i s)) 2) (c (w s) (i s)))
                                           (quotient (a (w s) (i s)) 2))
                                     (exp (i s) 2))
                              (plus (times (exp (sub1 (i s)) 2) (remainder (a (w s) (i s)) 2))
                                    (quotient (low (w s) (i s)) 2)))
                        (result s))))
 (wp (lambda (s) (wp-zcoef-g (f1 s)
                             (x s)
                             (c (w s) (i s))
                             (low (w s) (i s))
                             (a (w s) (i s))
                             (result s)
                             (f2 s)
                             (i s))))
 (measure (lambda (s) (dec (x s))))
 (sigma (lambda (s) (list (dec (x s))
                          (quotient (f1 s) 2)
                          (f2 s)
                          (result s)
                          (i s)
                          (if (equal (remainder (f1 s) 2) 0)
                              (quotient (w s) 2)
                            (plus (quotient (w s) 2) (times (f2 s) (exp (i s) 2)))))))
 (r (lambda (s) (and (not (zerop (x s)))
                     (not (lessp (i s) (x s)))
                     (lessp (f2 s) (exp (i s) 2))
                     (equal (plus (quotient (w s) (exp (x s) 2))
                                  (times (f2 s)
                                         (remainder (f1 s) (exp (sub1 (x s)) 2))
                                         (quotient (exp (i s) 2) (exp (sub1 (x s)) 2))))
                            (result s))))))
 ((disable-theory if-normalization)
  (hands-off difference)
  (expand (wp-zcoef-g (cadr s)
                      (car s)
                      (quotient (quotient (caddddddr s) (exp (caddddr s) 2))
                                (exp (caddddr s) 2))
                      (remainder (caddddddr s) (exp (caddddr s) 2))
                      (remainder (quotient (caddddddr s) (exp (caddddr s) 2))
                                 (exp (caddddr s) 2))
                      (cadddr s)
                      (caddr s)
                      (caddddr s))
          (exp x1 2))
  (disable quotient-add1-arg2
           remainder-add1-arg2
           remainder-plus-arg1
           remainder-plus-arg1-simple
           quotient-plus-arg1
           times
           times-add1
           equal-add1
           remainder-difference-arg1
           equal-transpose-meta
           lessp-transpose-meta
           lessp-2
           sub1-times
           sub1-quotient
           sub1-remainder
           remainder-minus-one-as-0
           lessp-quotient-arg2-linear
           lessp-quotient)))

(prove-lemma wp-zcoef-loop-invariant-flat (rewrite)
  (implies (and (not (zerop i))
                (numberp f1)
                (numberp low)
                (numberp a)
                (numberp c)
                (lessp low (exp i 2))
                (lessp a (exp i 2))
                (lessp f2 (exp i 2))
                (equal
                 (plus (quotient (plus low (times (exp i 2) a) (times (exp i 2) (exp i 2) c))
                                 (exp i 2))
                       (times f2
                              (remainder f1 (exp (sub1 i) 2))
                              (quotient (exp i 2) (exp (sub1 i) 2))))
                 result))
           (wp-zcoef-g f1 i c low a result f2 i))
  ((disable-theory if-normalization)
   (use (wp-zcoef-loop-invariant
           (s (list i f1 f2 result i
                   (plus low (times (exp i 2) a) (times (exp i 2) (exp i 2) c))))))
   (hands-off difference)
   (disable quotient-add1-arg2
              remainder-add1-arg2
              remainder-plus-arg1
              remainder-plus-arg1-simple
             quotient-plus-arg1
             remainder-difference-arg1
             times-add1
             times
             wp-zcoef-g)))

(prove-lemma wp-zcoef-is-correct (rewrite)
 (implies (and (lessp f2 256)
                   (lessp f1 256)
                   (numberp low)
                   (lessp low 256))
           (wp-zcoef-1 f1 c low f2))
  ((disable-theory if-normalization)
   (hands-off difference)
   (disable times
              times-add1
              quotient-add1-arg2
              remainder-add1-arg2)))
```

;;; Proof Using Tail Recursion Theory

```
(prove-lemma wp-zcoef-g-instance (rewrite)
   (equal (wp-zcoef f1 x c low a f1save f2)
           (wp-zcoef-g f1 x c low a (times f1save f2) f2 8))
   ((hands-off plus times quotient remainder difference)
    (expand (wp-zcoef f1 0 c low a f1save f2)
              (wp-zcoef-g f1 0 c low a (times f1save f2) f2 8))))

;;; Split the state into an "a" component that does the effective computation, and
;;; an "s" component that drives the computation.

;;; Package the "a" component into a long integer.

(defn c (a i) (quotient a (exp (times 2 i) 2)))                  ; the carry flag
(defn a (a i) (remainder (quotient a (exp i 2)) (exp i 2)))   ; the accumulator
(defn low (a i) (remainder a (exp i 2)))                          ; lower half of product
```

```
;;; Package the "s" component using lists.

(defn x (s) (car s))                          ; x register counter
(defn f1 (s) (cadr s))                        ; first factor
(defn f2 (s) (caddr s))                       ; second factor
(defn result (s) (cadddr s))                  ; the result
(defn i (s) (caddddr s))                      ; word size in bits

;;; Define the instantiation of h from the generic theory.  wp-zcoef-h performs a
;;; multiply in the standard way, except that it delivers twice the product.

(defn wp-zcoef-h (s)
  (if (equal (dec (x s)) 0)
      0
    (times 2 (plus (wp-zcoef-h (list (dec (x s))
                                     (quotient (f1 s) 2)
                                     (f2 s)
                                     (result s)
                                     (i s)))
                   (if (equal (remainder (f1 s) 2) 0)
                       0
                     (f2 s)))))
  ((lessp (dec (x s)))))

;;; Define the instantiation of hs.  btm-s computes the bottom object under tau.

(defn btm-s (s)
  (if (equal (dec (x s)) 0)
      s
    (btm-s (list (dec (x s))
                 (quotient (f1 s) 2)
                 (f2 s)
                 (result s)
                 (i s))))
  ((lessp (dec (x s)))))

;;; An alternative to the library rule remainder-plus-arg1 with fewer case splits.

(prove-lemma remainder-plus-arg1-alt (rewrite)
   (implies (equal (remainder a d) 0)
            (and (equal (remainder (plus a b) d) (remainder b d))
                 (equal (remainder (plus b a) d) (remainder b d))
                 (equal (remainder (plus b a c) d) (remainder (plus b c) d))
                 (equal (remainder (plus b c a) d) (remainder (plus b c) d))))
   ((disable-theory if-normalization)
    (hands-off difference)
    (disable times-add1)))
```

```
;;; This simple fact is absent from modularithmetic-98.

(prove-lemma equal-even-odd (rewrite)
            (implies (equal (remainder a 2) (remainder b 2))
                        (and (not (equal a (add1 b)))
                               (not (equal (add1 b) a))))
   ((disable-theory if-normalization)
    (disable remainder-add1-arg2)))

;;; Define an induction hint which recurses on two copies of wp-zcoef-g.

(defn ind-2 (f1 x c low a f1p xp cp lowp ap f2 i)
  (if (equal (dec x) 0)
      0
    (ind-2
     (plus (times (exp (sub1 i) 2) (remainder low 2)) (quotient f1 2))
     (dec x)
     (times (remainder f1 2)
              (quotient (plus (plus (times (exp (sub1 i) 2) c) (quotient a 2)) f2)
                           (exp i 2)))
     (plus (times (exp (sub1 i) 2) (remainder a 2)) (quotient low 2))
     (if (equal (remainder f1 2) 0)
         (plus (times (exp (sub1 i) 2) c) (quotient a 2))
               (remainder (plus (plus (times (exp (sub1 i) 2) c) (quotient a 2)) f2)
                               (exp i 2)))
     (plus (times (exp (sub1 i) 2) (remainder lowp 2)) (quotient f1p 2))
     (dec xp)
     (times (remainder f1p 2)
              (quotient (plus (plus (times (exp (sub1 i) 2) cp) (quotient ap 2)) f2)
                           (exp i 2)))
     (plus (times (exp (sub1 i) 2) (remainder ap 2)) (quotient lowp 2))
     (if (equal (remainder f1p 2) 0)
         (plus (times (exp (sub1 i) 2) cp) (quotient ap 2))
               (remainder (plus (plus (times (exp (sub1 i) 2) cp) (quotient ap 2)) f2)
                               (exp i 2)))
     f2
     i))
  ((lessp (dec x))))

;;; Backchain on the arguments of a functional equality.

(prove-lemma equal-wp-zcoef-g (rewrite)
   (implies (and (equal (remainder f1 (exp (sub1 x) 2)) (remainder f1p (exp (sub1 x) 2)))
                      (equal x xp)
                      (equal (plus (times c (exp i 2)) a) (plus (times cp (exp i 2)) ap))
                      (equal low lowp)
```

```
                    (not (lessp i x))
                    (not (zerop x)))
               (equal (equal (wp-zcoef-g f1 x c low a result f2 i)
                             (wp-zcoef-g f1p xp cp lowp ap result f2 i))
                      t))
   ((disable-theory if-normalization)
    (hands-off difference)
    (induct (ind-2 f1 x c low a f1p xp cp lowp ap f2 i))
    (expand (exp (sub1 xp) 2))
    (disable remainder-add1-arg2
             quotient-add1-arg2
             quotient-plus-arg1
             remainder-plus-arg1
             remainder-plus-arg1-simple
             times
             times-add1
             sub1-times
             sub1-plus
             sub1-remainder
             sub1-quotient
             remainder-times-arg1
             remainder-difference-arg1
             equal-add1
             remainder-minus-one-as-0
             equal-transpose-meta
             quotient-is-unique
             lessp-quotient-arg2-linear
             lessp-quotient
             plus-is-0
             remainder-add1-arg1
             equal-quotient-0
             quotient-times-arg1-simple
             quotient-add1-arg1)))

;;; An alternative to the library quotient-plus-arg1 with fewer case splits.

(prove-lemma quotient-plus-arg1-alt (rewrite)
   (implies (equal (remainder a d) 0)
            (and (equal (quotient (plus a b) d) (plus (quotient a d) (quotient b d)))
                 (equal (quotient (plus b a) d) (plus (quotient a d) (quotient b d)))
                 (equal (quotient (plus b a c) d) (plus (quotient a d) (quotient (plus b c) d)))
                (equal (quotient (plus b c a) d) (plus (quotient a d) (quotient (plus b c) d)))))
   ((disable-theory if-normalization)
    (hands-off difference)
    (disable lessp-transpose-meta
             equal-transpose-meta
             equal-even-odd
```

```
                remainder-difference-arg1
                remainder-plus-arg1)))

;;; A Knuth-Bendix style rule for remainder-times-arg2.

(prove-lemma remainder-times-arg2-kb (rewrite)
   (equal (plus (remainder a b) (times b (remainder (quotient a b) c)))
             (plus (remainder a c) (times c (remainder (quotient a c) b))))
  ((use (remainder-times-arg2 (a a) (zb b) (zc c))
          (remainder-times-arg2 (a a) (zb c) (zc b)))
    (disable remainder-times-arg2
                times
                quotient
                remainder)))

;;; A Knuth-Bendix style rule for quotient-times-arg2.

(prove-lemma remainder-times-arg2-close-kb (rewrite)
   (equal (plus (remainder (quotient a b) c) (times c (quotient (quotient a c) b)))
             (quotient a b)))

;;; More properties of exp.

(prove-lemma exp-plus-arg1 (rewrite)
   (equal (exp (plus i j) b) (times (exp i b) (exp j b))))

(prove-lemma remainder-exp-exp (rewrite)
   (implies (not (lessp i j))
                (equal (remainder (exp i 2) (exp j 2)) 0))
   ((disable times
                 times-add1)))

(prove-lemma quotient-exp-2 (rewrite)
   (equal (quotient (exp i 2) 2)
             (if (zerop i)
                 0
               (exp (sub1 i) 2))))

;;; Use the generic tail recursion theory.

(functionally-instantiate wp-zcoef-g=h nil
  (implies (and (not (zerop (x s)))
                      (not (lessp (i s) (x s)))
                      (numberp (f2 s))
                      (lessp (f2 s) (exp (i s) 2)))
             (equal (wp-zcoef-g (f1 s) (x s) (c a (i s)) (low a (i s)) (a a (i s)) (result s) (f2 s) (i s))
                        (if (equal (dec (x s)) 0)
                         (equal (plus (times (plus (times (exp (sub1 (i s)) 2) (c a (i s)))
                                                      (quotient (a a (i s)) 2))
                                             (exp (i s) 2))
                                     (plus (times (exp (sub1 (i s)) 2) (remainder (a a (i s)) 2))
                                             (quotient (low a (i s)) 2)))
                              (result s))
                         (let ((a (if (equal (dec (x s)) 0)
                                      a
                                     (quotient (plus a (times (exp (i s) 2) (wp-zcoef-h s)))
                                                  (exp (sub1 (x s)) 2))))
                                (s (btm-s s)))
                            (equal (plus (times (plus (times (exp (sub1 (i s)) 2) (c a (i s)))
                                                        (quotient (a a (i s)) 2))
                                               (exp (i s) 2))
                                        (plus (times (exp (sub1 (i s)) 2) (remainder (a a (i s)) 2))
                                               (quotient (low a (i s)) 2)))
                                  (result s))))))
g=h
((bb (lambda (s) (equal (dec (x s)) 0)))
 (qt (lambda (a s) (equal (plus (times (plus (times (exp (sub1 (i s)) 2) (c a (i s)))
                                                    (quotient (a a (i s)) 2))
                                           (exp (i s) 2))
                                    (plus (times (exp (sub1 (i s)) 2) (remainder (a a (i s)) 2))
                                           (quotient (low a (i s)) 2)))
                              (result s))))
 (g (lambda (a s)
       (wp-zcoef-g (f1 s) (x s) (c a (i s)) (low a (i s)) (a a (i s)) (result s) (f2 s) (i s))))
 (measure-g (lambda (s) (dec (x s))))
 (tau (lambda (s) (list (dec (x s))
                         (quotient (f1 s) 2)
                         (f2 s)
                         (result s)
                         (i s))))
 (rho (lambda (a s)
       (if (equal (remainder (f1 s) 2) 0)
          (quotient a 2)
         (plus (quotient a 2) (times (f2 s) (exp (i s) 2))))))
 (rhoh (lambda (a s) (if (equal (remainder (f1 s) 2) 0)
                          (times 2 a)
                         (times 2 (plus a (f2 s))))))
 (h (lambda (s) (wp-zcoef-h s)))
 (rt (lambda (a s) (and (not (zerop (x s)))
                         (not (lessp (i s) (x s)))
                         (numberp (f2 s))
                         (lessp (f2 s) (exp (i s) 2)))))
 (id (lambda () 0))
 (op (lambda (a x s) (if (equal (dec (x s)) 0)
                                  a
                                 (quotient (plus a (times (exp (i s) 2) x)) (exp (sub1 (x s)) 2)))))
 (hs (lambda (s) (btm-s s))))
((disable-theory if-normalization)
 (hands-off difference)
 (expand (times 2 x)
         (times 2 z))
 (disable quotient-add1-arg2
         remainder-add1-arg2
         remainder-plus-arg1
         remainder-plus-arg1-simple
         quotient-plus-arg1
         remainder-difference-arg1
         times-add1
         times
         equal-add1
         equal-transpose-meta
         lessp-transpose-meta
         lessp-2
         quotient-plus-arg2
         lessp-quotient-arg2-linear
         remainder-add1-arg1
         sub1-remainder
         sub1-quotient
         sub1-times
         sub1-plus
         quotient-remainder
         remainder-minus-one-as-0
         lessp-quotient
         lessp-odometer-simple
         equal-odometer-simple
         plus-is-0
         no-divisors-of-zero)))

;;; Since the bottom object under tau was defined recursively this is needed.

(prove-lemma btm-s-destruct (rewrite)
 (implies (and (not (zerop (x s)))
               (not (lessp (i s) (x s))))
         (and (equal (car (btm-s s)) 1)
              (equal (cadr (btm-s s)) (if (equal (dec (x s)) 0)
                                          (f1 s)
                                         (quotient (f1 s) (exp (sub1 (x s)) 2))))
              (equal (caddr (btm-s s)) (f2 s))
              (equal (cadddr (btm-s s)) (result s))
              (equal (caddddr (btm-s s)) (i s))))
 ((disable-theory if-normalization)
  (hands-off difference)
  (disable quotient-add1-arg2
           remainder-add1-arg2
           quotient-sub1-arg1
           times
           times-add1)))

;;; Another Knuth-Bendex style rule for quotient-times-arg2.

(prove-lemma quotient-times-arg2-permutative (rewrite)
  (implies (equal (remainder c b) 0)
           (and (equal (quotient (quotient (plus a c) d) b)
                       (quotient (plus (quotient a b) (quotient c b)) d))
                (equal (quotient (quotient (quotient (plus a c) d) e) b)
                       (quotient (quotient (plus (quotient a b) (quotient c b)) d) e))))
 ((disable-theory if-normalization)
  (hands-off difference)
  (use (quotient-times-arg2-kb (a (plus a c)) (b d) (c b))
       (quotient-times-arg2-kb (a (quotient (plus a c) d)) (b e) (c b))
       (quotient-times-arg2-kb (a a) (b b) (c d))
       (quotient-times-arg2-kb (a c) (b b) (c d))
       (quotient-plus-arg1 (a a) (b c) (c b)))
  (disable quotient-times-arg2-kb
           quotient-plus-arg1
           remainder
           quotient
           plus
           lessp-times-single-linear
           lessp-quotient-arg2-linear
           quotient-plus-arg2
           lessp-quotient
           lessp-transpose-meta
           equal-transpose-meta
           quotient-add1-arg1
           equal-even-odd
           sub1-plus
           remainder-add1-arg2)))

(prove-lemma remainder-exp-2 (rewrite)
  (equal (remainder (exp i 2) 2)
         (if (zerop i) 1 0))
  ((expand (exp i 2))))

(prove-lemma equal-exp-0 (rewrite)
  (equal (equal (exp i b) 0)
         (zerop b)))
```

```
;;; Avoids an expand hint.

(prove-lemma quotient-is-0 (rewrite)
   (implies (lessp a b) (equal (quotient a b) 0)))

;;; Convert generic theory back to flat state space.

(prove-lemma wp-zcoef-g=h-flat (rewrite)
  (implies (and (not (zerop i))
                      (numberp f2)
                      (lessp f2 (exp i 2))
                      (numberp a)
                      (lessp a (exp i 2))
                      (numberp low)
                      (lessp low (exp i 2)))
              (equal (wp-zcoef-g f1 i 0 low a result f2 i)
                        (if (equal (dec i) 0)
                            (equal a result)
                          (equal (plus a (wp-zcoef-h (list i f1 f2 result i)))
                                    result))))
  ((disable-theory if-normalization)
   (use (wp-zcoef-g=h
            (a (plus low (times (exp i 2) a)))
            (s (list i f1 f2 result i))))
   (hands-off difference)
   (expand (times 2 i))
   (disable quotient-add1-arg2
               remainder-add1-arg2
               remainder-plus-arg1
               remainder-plus-arg1-simple
               quotient-plus-arg1
               remainder-difference-arg1
               times-add1
               times
               wp-zcoef-g
               wp-zcoef-h
               sub1-times
               sub1-plus
               sub1-remainder
               sub1-quotient
               quotient-times-arg1
               lessp-transpose-meta
               lessp-quotient-arg2-linear)))

;;; Prove the primitive recursive version multiplies.

(prove-lemma wp-zcoef-h-multiplies (rewrite)
  (implies (and (not (zerop (x s)))
                      (not (lessp (i s) (x s))))
                (equal (wp-zcoef-h s)
                         (times 2 (f2 s) (remainder (f1 s) (exp (sub1 (x s)) 2)))))
  ((disable-theory if-normalization)
   (disable quotient-add1-arg2
               remainder-add1-arg2
               quotient-sub1-arg1
               times
               times-add1
               difference-add1
               sub1-quotient
               remainder-difference-arg1
               sub1-times)))

;;; The program correctness proof.

(prove-lemma wp-zcoef-is-correct (rewrite)
  (implies (and (numberp f2)
                      (lessp f2 256)
                      (numberp f1)
                      (lessp f1 256)
                      (numberp low)
                      (lessp low 256))
               (wp-zcoef-1 f1 c low f2))
  ((disable-theory if-normalization)
   (hands-off difference)
   (disable times
               times-add1
               quotient-add1-arg2
               remainder-add1-arg2
               equal-add1)))
```

;;; Proof Using the Alternative Induction Theory

```
(prove-lemma wp-zcoef-g-instance (rewrite)
   (equal (wp-zcoef f1 x c low a f1save f2)
            (wp-zcoef-g f1 x c low a (times f1save f2) f2 8))
   ((disable-theory if-normalization)
    (expand (wp-zcoef f1 0 c low a f1save f2)
              (wp-zcoef-g f1 0 c low a (times f1save f2) f2 8))
    (hands-off quotient remainder difference)
    (disable times
              times-add1)))

;;; Inspection of the assembly program shows that c effectively behaves as an
;;; extension of a, so we define an equivalent function on ac = a + c*2^i.
```

```
(defn wp-zcoef-ac (f1 x ac low result f2 i)
  (if (equal (dec x) 0)
      (equal (plus (quotient low 2) (times ac (exp (sub1 i) 2)))
             result)
    (wp-zcoef-ac
     (plus (quotient f1 2) (times (exp (sub1 i) 2) (remainder low 2)))
     (dec x)
     (if (equal (remainder f1 2) 0)
         (quotient ac 2)
       (plus f2 (quotient ac 2)))
     (plus (quotient low 2) (times (exp (sub1 i) 2) (remainder ac 2)))
     result
     f2
     i))
  ((lessp (dec x))))

;;; We include all variable in wp-zcoef-g and wp-zcoef-ac within a common state.

(defn f1 (s) (car s))
(defn x (s) (cadr s))
(defn c (s) (caddr s))
(defn low (s) (cadddr s))
(defn a (s) (car (cddddr s)))
(defn result (s) (cadr (cddddr s)))
(defn f2 (s) (caddr (cddddr s)))
(defn i (s) (cadddr (cddddr s)))
(defn ac (s) (car (cddddr (cddddr s))))

;;; An alternative to remainder-plus-arg1 that generates fewer case splits.

(prove-lemma remainder-plus-arg1-alt (rewrite)
   (implies (equal (remainder a d) 0)
            (and (equal (remainder (plus a b) d) (remainder b d))
                 (equal (remainder (plus b a) d) (remainder b d))
                 (equal (remainder (plus b a c) d) (remainder (plus b c) d))
                 (equal (remainder (plus b c a) d) (remainder (plus b c) d))))
  ((disable-theory if-normalization)
    (hands-off difference)
    (disable times-add1)))

;;; An alternative to quotient-plus-arg1 that generates fewer case splits.

(prove-lemma quotient-plus-arg1-alt (rewrite)
   (implies (equal (remainder a d) 0)
             (and (equal (quotient (plus a b) d) (plus (quotient a d) (quotient b d)))
                  (equal (quotient (plus b a) d) (plus (quotient a d) (quotient b d)))
```

```
                        (equal (quotient (plus b a c) d) (plus (quotient a d) (quotient (plus b c) d)))
                       (equal (quotient (plus b c a) d) (plus (quotient a d) (quotient (plus b c) d)))))
  ((disable-theory if-normalization)
   (hands-off difference)
   (disable lessp-transpose-meta
            equal-transpose-meta
            remainder-difference-arg1
            remainder-plus-arg1)))

;;; We now prove an instance of fn1-as-fn2.

(functionally-instantiate wp-zcoef-g-as-ac nil
  (let ((ts (list (f1 s)
                  (x s)
                  (c s)
                  (low s)
                  (a s)
                  (result s)
                  (f2 s)
                  (i s)
                  (plus (a s) (times (c s) (exp (i s) 2))))))
    (implies (and (not (zerop (x s)))
                  (not (lessp (i s) (x s))))
             (equal (wp-zcoef-g (f1 s) (x s) (c s) (low s) (a s) (result s) (f2 s) (i s))
                    (wp-zcoef-ac (f1 ts) (x ts) (ac ts) (low ts) (result ts) (f2 ts) (i ts))))))
    fn1-as-fn2
    ((b1 (lambda (s) (equal (dec (x s)) 0)))
     (b2 (lambda (s) (equal (dec (x s)) 0)))
     (q1 (lambda (s) (equal (plus (times (plus (times (exp (sub1 (i s)) 2) (c s))
                                               (quotient (a s) 2))
                                         (exp (i s) 2))
                                  (plus (times (exp (sub1 (i s)) 2) (remainder (a s) 2))
                                        (quotient (low s) 2)))
                            (result s))))
     (q2 (lambda (s) (equal (plus (quotient (low s) 2) (times (ac s) (exp (sub1 (i s)) 2)))
                            (result s))))
     (p (lambda (s) (and (not (zerop (x s)))
                         (not (lessp (i s) (x s))))))
     (fn1 (lambda (s)
            (wp-zcoef-g (f1 s) (x s) (c s) (low s) (a s) (result s) (f2 s) (i s))))
     (fn2 (lambda (s)
            (wp-zcoef-ac (f1 s) (x s) (ac s) (low s) (result s) (f2 s) (i s))))
     (sigma1 (lambda (s)
               (list (plus (times (exp (sub1 (i s)) 2) (remainder (low s) 2))
                           (quotient (f1 s) 2))
                     (dec (x s))
                     (if (equal (remainder (f1 s) 2) 0)
                              0
                             (quotient (plus (plus (times (exp (sub1 (i s)) 2) (c s))
                                                   (quotient (a s) 2))
                                             (f2 s))
                                       (exp (i s) 2)))
                            (plus (times (exp (sub1 (i s)) 2) (remainder (a s) 2))
                                  (quotient (low s) 2))
                            (if (equal (remainder (f1 s) 2) 0)
                               (plus (times (exp (sub1 (i s)) 2) (c s)) (quotient (a s) 2))
                             (remainder (plus (plus (times (exp (sub1 (i s)) 2) (c s))
                                                    (quotient (a s) 2))
                                              (f2 s))
                                        (exp (i s) 2)))
                            (result s)
                            (f2 s)
                            (i s))))
   (sigma2 (lambda (s)
                 (list (plus (times (exp (sub1 (i s)) 2) (remainder (low s) 2))
                             (quotient (f1 s) 2))
                       (dec (x s))
                       (if (equal (remainder (f1 s) 2) 0)
                            0
                         (quotient (plus (plus (times (exp (sub1 (i s)) 2) (c s))
                                               (quotient (a s) 2))
                                         (f2 s))
                                   (exp (i s) 2)))
                       (plus (times (exp (sub1 (i s)) 2) (remainder (ac s) 2))
                             (quotient (low s) 2))
                       (if (equal (remainder (f1 s) 2) 0)
                          (plus (times (exp (sub1 (i s)) 2) (c s)) (quotient (a s) 2))
                        (remainder (plus (plus (times (exp (sub1 (i s)) 2) (c s))
                                               (quotient (a s) 2))
                                         (f2 s))
                                   (exp (i s) 2)))
                       (result s)
                       (f2 s)
                       (i s)
                       (if (equal (remainder (f1 s) 2) 0)
                          (quotient (ac s) 2)
                        (plus (f2 s) (quotient (ac s) 2))))))
   (measure1 (lambda (s) (if (zerop (x s)) 256 (x s))))
   (id-alt (lambda (s) (list (f1 s) (x s) (c s) (low s) (a s) (result s) (f2 s) (i s)
                             (plus (a s) (times (c s) (exp (i s) 2)))))))
 ((disable-theory if-normalization)
  (expand (wp-zcoef-g (car s)
                      (cadr s)
                      (caddr s)
```

```
                    (cadddr s)
                    (caddddr s)
                    (cadddddr s)
                    (caddddddr s)
                    (cadddddddr s))
            (wp-zcoef-g (car s)
                    0
                    (caddr s)
                    (cadddr s)
                    (caddddr s)
                    (cadddddr s)
                    (caddddddr s)
                    (cadddddddr s)))
  (hands-off difference)
  (disable quotient-add1-arg2
           remainder-add1-arg2
           quotient-plus-arg1
           remainder-plus-arg1
           remainder-plus-arg1-simple
           quotient-times-arg1
           remainder-times-arg1
           lessp-transpose-meta
           equal-transpose-meta
           times
           times-add1
           sub1-times
           equal-add1
           remainder-difference-arg1
           sub1-quotient
           sub1-remainder
           wp-zcoef-g)))

;;; Convert the above instantiation into an effective rewrite rule.

(prove-lemma wp-zcoef-g-as-ac-rewrite (rewrite)
  (implies (and (not (zerop x))
                (not (lessp i x)))
           (equal (wp-zcoef-g f1 x c low a result f2 i)
                  (wp-zcoef-ac f1
                               x
                               (plus a (times c (exp i 2)))
                               low
                               result
                               f2
                               i)))
  ((use (wp-zcoef-g-as-ac (s (list f1 x c low a result f2 i
                                 (plus a (times c (exp i 2)))))))))
```

```
    (hands-off quotient remainder times difference plus)))

(prove-lemma remainder-exp-exp (rewrite)
   (implies (not (lessp i j))
            (equal (remainder (exp i 2) (exp j 2)) 0))
   ((disable times
             times-add1)))

;;; We now look for a substitution id-alt, which leaves wp-zcoef-ac invariant.
;;; Since we will be proving that sigma1 and id-alt commute, we would indeed have
;;; a simple proof if sigma1 and id-alt altered disjoint sets of variables.  Only
;;; f2, result and i are left unchanged by sigma1.  Looking at the assembly
;;; language program, we see that a change in f2 only affects ac.  So if we
;;; decremented f2 and incremented ac whenever f1 = 1 mod 2, the computation
;;; would be unchanged.  Notice that a single change in f2 could effect the
;;; computation on the next x - 1 iterations.  So it is necessary to add
;;; (times 2 (remainder (exp (sub1 x) 2))) to ac.  This defines id-alt.

(functionally-instantiate f2-induction nil
 (let ((ts
        (list (f1 s)
              (x s)
              (c s)
              (low s)
              (a s)
              (result s)
              (sub1 (f2 s))
              (i s)
              (plus (ac s) (times 2 (remainder (f1 s) (exp (sub1 (x s)) 2)))))))
   (implies
    (and (not (zerop (f2 s)))
         (not (zerop (x s)))
         (not (lessp (i s) (x s))))
    (equal (wp-zcoef-ac (f1 s) (x s) (ac s) (low s) (result s) (f2 s) (i s))
           (wp-zcoef-ac (f1 ts) (x ts) (ac ts) (low ts) (result ts) (f2 ts) (i ts)))))
    fn1-as-fn2
    ((b1 (lambda (s) (equal (dec (x s)) 0)))
     (b2 (lambda (s) (equal (dec (x s)) 0)))
     (q1 (lambda (s) (equal (plus (quotient (low s) 2)
                                  (times (ac s) (exp (sub1 (i s)) 2)))
                            (result s))))
     (q2 (lambda (s) (equal (plus (quotient (low s) 2)
                                  (times (ac s) (exp (sub1 (i s)) 2)))
                            (result s))))
     (p (lambda (s) (and (not (zerop (f2 s)))
                         (not (zerop (x s)))
                         (not (lessp (i s) (x s))))))
    (fn1 (lambda (s) (wp-zcoef-ac (f1 s) (x s) (ac s) (low s) (result s) (f2 s) (i s))))
    (fn2 (lambda (s) (wp-zcoef-ac (f1 s) (x s) (ac s) (low s) (result s) (f2 s) (i s))))
    (sigma1 (lambda (s)
                (list (plus (times (exp (sub1 (i s)) 2) (remainder (low s) 2))
                            (quotient (f1 s) 2))
                      (dec (x s))
                      (c s)
                      (plus (times (exp (sub1 (i s)) 2) (remainder (ac s) 2))
                            (quotient (low s) 2))
                      (a s)
                      (result s)
                      (f2 s)
                      (i s)
                      (if (equal (remainder (f1 s) 2) 0)
                          (quotient (ac s) 2)
                        (plus (f2 s) (quotient (ac s) 2))))))
    (sigma2 (lambda (s)
                 (list (plus (times (exp (sub1 (i s)) 2) (remainder (low s) 2))
                             (quotient (f1 s) 2))
                      (dec (x s))
                      (c s)
                      (plus (times (exp (sub1 (i s)) 2) (remainder (ac s) 2))
                            (quotient (low s) 2))
                      (a s)
                      (result s)
                      (f2 s)
                      (i s)
                      (if (equal (remainder (f1 s) 2) 0)
                          (quotient (ac s) 2)
                       (plus (f2 s) (quotient (ac s) 2))))))
    (measure1 (lambda (s) (if (zerop (x s)) 256 (x s))))
    (id-alt (lambda (s) (list (f1 s) (x s) (c s) (low s) (a s) (result s) (sub1 (f2 s)) (i s)
                              (plus (ac s) (times 2 (remainder (f1 s) (exp (sub1 (x s)) 2))))))))
   ((disable-theory if-normalization)
    (hands-off difference)
    (disable quotient-add1-arg2
             remainder-add1-arg2
             remainder-difference-arg1
             quotient-plus-arg1
             remainder-plus-arg1
             remainder-plus-arg1-simple
             times
             times-add1)))

;;; Convert the above to an effective rewrite rule.

(prove-lemma f2-induction-rewrite (rewrite)
  (implies (and (not (zerop f2))
                (not (zerop x))
                (not (lessp i x)))
           (equal (wp-zcoef-ac f1 x ac low result f2 i)
                  (wp-zcoef-ac f1
                               x
                               (plus ac (times 2 (remainder f1 (exp (sub1 x) 2))))
                               low
                               result
                               (sub1 f2)
                               i)))
  ((use (f2-induction (s (list f1 x c low a result f2 i ac))))
   (hands-off quotient remainder times difference plus)))

;;; This is the base case for the alternative induction.

(prove-lemma f2-induction-base-case (rewrite)
  (implies (and (zerop f2)
                (not (zerop x))
                (not (lessp i x)))
           (equal (wp-zcoef-ac f1 x ac low result f2 i)
                  (equal (plus (quotient low (exp x 2))
                               (times ac (quotient (exp i 2) (exp x 2))))
                         result)))
  ((disable-theory if-normalization)
   (induct (wp-zcoef-ac f1 x ac low result f2 i))
   (hands-off difference)
   (expand (wp-zcoef-ac f1 x ac low result f2 i)
           (wp-zcoef-ac f1 x ac low result 0 i))
   (disable quotient-add1-arg2
           remainder-add1-arg2
           quotient-plus-arg1
           remainder-plus-arg1
           remainder-plus-arg1-simple
           remainder-difference-arg1
           times
           times-add1
           equal-add1
           wp-zcoef-ac
           lessp-transpose-meta
           equal-transpose-meta
           remainder-remainder
           sub1-quotient
           lessp-odometer-simple
           equal-odometer-simple
           sub1-times
           sub1-remainder)))
```

```
;;; This is the induction hint corresponding to id-alt.

(defn wp-ind (f1 x ac f2)
  (if (zerop f2)
      t
    (wp-ind f1 x (plus ac (times 2 (remainder f1 (exp (sub1 x) 2)))) (sub1 f2))))

;;; Rewrite an arbitrary application of wp-zcoef-ac into the base case.

(prove-lemma wp-zcoef-ac-as-0 (rewrite)
   (implies (and (not (zerop x))
                 (not (lessp i x)))
            (equal (wp-zcoef-ac f1 x ac low result f2 i)
                   (if (zerop f2)
                     (equal (plus (quotient low (exp x 2))
                                  (times ac (quotient (exp i 2) (exp x 2))))
                            result)
                    (wp-zcoef-ac f1
                                 x
                                 (plus ac (times 2 f2 (remainder f1 (exp (sub1 x) 2))))
                                 low
                                 result
                                 0
                                 i))))
  ((disable-theory if-normalization)
   (induct (wp-ind f1 x ac f2))
   (hands-off difference)
   (disable wp-zcoef-ac
            plus-commutes-nest-meta
            plus-commutes-meta
            quotient-plus-arg2)))

;;; Finally, the correctness result.

(prove-lemma mult-program-is-correct nil
   (implies (and (lessp low 256)
                 (lessp f1 256)
                 (lessp f2 256))
            (wp-zcoef-1 f1 c low f2))
  ((disable-theory if-normalization)
   (disable quotient-add1-arg2
            remainder-add1-arg2
            times
            times-add1
            quotient-sub1-arg1
            difference-add1)))
```