Mechanical Verification of Register-Transfer Logic

David M. Russinoff
Advanced Micro Devices, Inc.

`david.russinoff@amd.com`
`http://www.onr.com/user/russ/david`

Outline

- An ACL2 library of floating-point arithmetic
  - Bit vectors
  - Logical operations
  - Floating-point representation
  - Rounding
  - Examples
- Application: a floating-point multiplier
  - AMD RTL language
  - Pipeline circuits
  - RTL model of the multiplier
  - Statement of IEEE compliance
  - Translation to ACL2
  - Proof of correctness

Bit Vectors

For natural numbers x, i, j, we define

$$x[i] = rem(\lfloor x/2^i \rfloor, 2)$$

and

$$x[i:j] = \lfloor rem(x, 2^{i+1})/2^j \rfloor.$$

These definitions are formalized as follows:

```
(DEFUN BITN (X K) (IF (LOGBITP X K) 1 0))

(DEFUN FL (X) (FLOOR X 1))

(DEFTHM BITN-DEF
    (IMPLIES (AND (INTEGERP X)
                  (INTEGERP I)
                  (>= X 0)
                  (>= I 0))
             (EQUAL (BITN X I)
                    (REM (FL (/ X (EXPT 2 I)))
                         2)))
  :RULE-CLASSES ())

(DEFUN BITS (X I J)
  (FL (/ (REM X (EXPT 2 (1+ I))) (EXPT 2 J))))
```


Logical Operations

The binary logical operations are defined recursively. For example,

$$x \mathbin{\texttt{\&}} y = \begin{cases} 0 & \text{if } x = 0 \\ 2(\lfloor x/2 \rfloor \mathbin{\texttt{\&}} \lfloor y/2 \rfloor) + 1 & \text{if } xy \text{ is odd} \\ 2(\lfloor x/2 \rfloor \mathbin{\texttt{\&}} \lfloor y/2 \rfloor) & \text{otherwise.} \end{cases}$$

The definitions of $x \mathbin{\texttt{|}} y$ and $x \mathbin{\texttt{\^{}}} y$ are similar.

If $x < 2^n$, then the *n-bit complement* of x is

$$comp1(x, n) = 2^n - x - 1.$$

Notation: $\texttt{\~{}}x[i:j] = comp1(x[i:j], i-j+1)$.

```
(DEFTHM LOGAND-DEF
    (IMPLIES (AND (INTEGERP X) (>= X 0)
                  (INTEGERP Y) (>= Y 0))
             (= (LOGAND X Y)
                (+ (* 2
                      (LOGAND (FL (/ X 2))
                              (FL (/ Y 2))))
                   (LOGAND (REM X 2) (REM Y 2)))))
  :RULE-CLASSES ())

(DEFUN COMP1 (X N)
  (1- (- (EXPT 2 N) X)))
```

Floating-Point Numbers

Every $r \in \mathbb{R}^*$ admits a unique factorization

$$r = sgn(r)sig(r)2^{expo(r)},$$

where

$sgn(r) = \pm 1$, $1 \leq sig(r) < 2$, and $expo(r) \in \mathbb{Z}$.

```
(DEFUN EXPO-MEASURE (X)
  (COND ((NOT (RATIONALP X)) 0)
        ((< X 0) '(2 . 0))
        ((< X 1) '(1 . 0))
        (T (FL X))))

(DEFUN EXPO (X)
  (DECLARE (XARGS :MEASURE (EXPO-MEASURE X)))
  (COND ((OR (NOT (RATIONALP X)) (= X 0)) ())
        ((< X 0) (EXPO (- X)))
        ((< X 1) (- (EXPO (/ X))))
        ((< X 2) 0)
        (T (1+ (EXPO (/ X 2))))))

(DEFUN SGN (X) (IF (< X 0) -1 +1))

(DEFUN SIG (X)
  (IF (< X 0)
      (- (* X (EXPT 2 (- (EXPO X)))))
    (* X (EXPT 2 (- (EXPO X))))))
```


Floating-Point Representations

A *floating-point format* is a pair of positive integers, representing the number of bits allocated for *sig* and *expo*. With respect to the $\phi = (p, q)$ format, the bit vector

$$x = \boxed{\sigma}\boxed{e_1}\boxed{\cdots}\boxed{e_q}\boxed{s_1}\boxed{\cdots}\boxed{s_p}$$

where we assume $s_1 = 1$ (normal representation), represents the rational number

$$decode(x, \phi) = r = sgn(r)sig(r)2^{expo(r)},$$

where

$$sgn(r) = (-1)^{\sigma},$$

$$sig(r) = (s_1.s_2...s_p)_2,$$

$$expo(r) = (e_1 \ldots e_q)_2 - (2^{q-1} - 1).$$

Thus $r \in \mathbb{R}^*$ is ϕ-representable iff

(a) $1 - 2^{q-1} \leq expo(r) \leq 2^{q-1}$, and

(b) $2^{p-1}sig(r) \in \mathbb{Z}$.

We shall say that r is *p-exact* if $2^{p-1}sig(r) \in \mathbb{Z}$.

Rounding

A *rounding mode* is a function $\mathcal{M}$ that assigns an n-exact number $\mathcal{M}(x, n)$ to every real number x and positive integer n.

The IEEE standard supports four rounding modes:

(1) *trunc* (round toward 0)

(2) *inf* (round toward ∞)

(3) *minf* (round toward $-\infty$)

(4) *near* (round to nearest)

For example,

$$trunc(x, n) = sgn(x)\lfloor 2^{n-1}sig(x)\rfloor 2^{expo(x)-(n-1)}.$$

Thus,

$$\begin{aligned} trunc((10101.1)_2, 3) &= \lfloor 2^2 \cdot (1.01011)_2 \rfloor \cdot 2^{4-2} \\ &= \lfloor (101.011)_2 \rfloor \cdot 2^2 \\ &= (101)_2 \cdot 2^2 \\ &= (10100)_2. \end{aligned}$$

Rounding: a Typical Lemma

Lemma *If x is $[k + expo(x) - expo(y)]$-exact, then*

$$x + trunc(y, k) = trunc(x + y, k + expo(x + y) - expo(y)).$$

$$\begin{array}{r} \overbrace{\texttt{1x}\overbrace{\texttt{xxxx.xxxxxxxx}}^{k}}^{k+expo(x)-expo(y)} \\ \texttt{1yyy.yyyyyyyy}\cdots \\ \hline \underbrace{\texttt{1zzzzzz.zzzzzzzz}}_{k+expo(x+y)-expo(y)}\cdots \end{array}$$

Proof: Let

$$k' = k + expo(x) - expo(y),$$

$$k'' = k + expo(x + y) - expo(y).$$

Since x is k'-exact,

$$x2^{k'-1-expo(x)} = x2^{k-1-expo(y)} \in \mathbb{Z}$$

and

$$\begin{aligned} x + trunc(y, k) &= x + \lfloor 2^{k-1-expo(y)}y \rfloor 2^{expo(y)+1-k} \\ &= (x2^{k-1-expo(y)} + \lfloor 2^{k-1-expo(y)}y \rfloor)2^{expo(y)+1-k} \\ &= \lfloor 2^{k-1-expo(y)}(x + y) \rfloor 2^{expo(y)+1-k} \\ &= \lfloor 2^{k''-1-expo(x+y)}(x + y) \rfloor 2^{expo(x+y)+1-k''} \\ &= trunc(x + y, k''). \square \end{aligned}$$

Bit Vectors: a Typical Lemma

Lemma *Let $x, n, k \in \mathbb{N}$. If $k < n$ and $x < 2^n$, then*

$$x \,\&\, (2^n - 2^k) = 2^k(x[n-1:k]).$$

Proof: For $k = 0$, we show by induction that

$$x \,\&\, (2^n - 1) = x.$$

The case $n = 0$ is trivial, and for $n > 0$, since $\lfloor (2^n - 1)/2 \rfloor = 2^{n-1} - 1$, we have

$$\begin{aligned} x \,\&\, (2^n - 1) &= 2(\lfloor x/2 \rfloor \,\&\, (2^{n-1} - 1)) + rem(x, 2) \\ &= 2\lfloor x/2 \rfloor + rem(x, 2) \\ &= x. \end{aligned}$$

Now, for $k > 0$,

$$\begin{aligned} x \,\&\, (2^n - 2^k) &= 2(\lfloor x/2 \rfloor \,\&\, (2^{n-1} - 2^{k-1})) \\ &= 2 \cdot 2^{k-1} \lfloor x/2 \rfloor [n-2:k-1] \\ &= 2^k \lfloor rem(\lfloor x/2 \rfloor, 2^{n-1})/2^{k-1} \rfloor \\ &= 2^k \lfloor \lfloor x/2 \rfloor / 2^{k-1} \rfloor \\ &= 2^k \lfloor x/2^k \rfloor \\ &= 2^k(x[n-1:k]).\square \end{aligned}$$

Implementation of Rounding

Lemma *Let x, n, and p be integers such that $0 < p < n$ and $2^{n-1} \leq x < 2^n$, and let $\mathcal{M}$ be an IEEE rounding mode. Then*

$$\mathcal{M}(x, p) = (x + \mathcal{C}) \,\&\, \left(2^n - 2^{n-\nu}\right)$$

where $\mathcal{C} = \begin{cases} 2^{n-p-1} & \text{if } \mathcal{M} = near \\ 2^{n-p} - 1 & \text{if } \mathcal{M} = inf \\ 0 & \text{if } \mathcal{M} = trunc \text{ or } \mathcal{M} = minf \end{cases}$

and

$\nu = \begin{cases} p - 1 & \text{if } \mathcal{M} = near,\ x \text{ is } (p+1)\text{-exact, not } p\text{-exact} \\ p & \text{otherwise.} \end{cases}$

Proof for the case $\mathcal{M} = trunc$:

$$\begin{aligned} trunc(x, p) &= \lfloor 2^{p-1-expo(x)} x \rfloor 2^{expo(x)+1-p} \\ &= \lfloor x/2^{n-p} \rfloor 2^{n-p} \\ &= 2^{n-p}(x[n-1:n-p]) \\ &= x \,\&\, (2^n - 2^{n-p}).\square \end{aligned}$$

Leading One Prediction

Given distinct n-bit integers a and b, can we compute in constant time (independent of n) an integer λ such that

$$expo(a - b) = expo(\lambda)?$$

No, but:

Lemma: *Given $a, b \in \mathbb{N}$ with $a \neq b$ and $expo(a) = expo(b) = n - 1$, let λ be defined as follows:*

$$\begin{aligned} \lambda_t &= a[n-1:0] \;\hat{}\; \tilde{}b[n-1:0] \\ \lambda_a &= a[n-1:0] \,\&\, \tilde{}b[n-1:0] \\ \lambda_b &= \tilde{}a[n-1:0] \,\&\, b[n-1:0] \\ \lambda_0 &= (\lambda_t[n:2] \,\&\, \lambda_a[n-2:1] \,\&\, \tilde{}\lambda_b[n-3:0]) \mid \\ &\quad\ (\tilde{}\lambda_t[n:2] \,\&\, \lambda_b[n-2:1] \,\&\, \tilde{}\lambda_b[n-3:0]) \mid \\ &\quad\ (\lambda_t[n:2] \,\&\, \lambda_b[n-2:1] \,\&\, \tilde{}\lambda_a[n-3:0]) \mid \\ &\quad\ (\tilde{}\lambda_t[n:2] \,\&\, \lambda_a[n-2:1] \,\&\, \tilde{}\lambda_a[n-3:0]) \\ \lambda &= 2\lambda_0 \;\hat{}\; \lambda_t[0] \end{aligned}$$

Then

$$expo(a - b) \in \{expo(\lambda), expo(\lambda) - 1\}.$$

$$\begin{aligned} a &= \texttt{1 0 1 1 0 1 1 0 0 0 1 0 1 0 0 \ldots 0} \\ b &= \texttt{1 0 1 1 0 1 0 1 1 1 1 1 0 0 0 \ldots 0} \\ a-b &= \texttt{0 0 0 0 0 0 0 0 0 0 1 1 1 0 0 \ldots 0} \\ \lambda &= \texttt{0 0 0 0 0 0 0 0 0 1 0 0 1 0 0 \ldots 0} \\ \sigma &= \texttt{T T T T T T A B B B T B A T T \ldots T} \end{aligned}$$

Let $\sigma = \sigma_{n-1} \cdots \sigma_0 \in \{\mathtt{T}, \mathtt{A}, \mathtt{B}\}^n$ be defined by

$$\sigma_i = \begin{cases} \mathtt{T} & \text{if } a_i = b_i \\ \mathtt{A} & \text{if } a_i > b_i \\ \mathtt{B} & \text{if } a_i < b_i. \end{cases}$$

Let $\sigma_{n-1} \cdots \sigma_\ell$ be the maximal prefix of σ of the form $\mathtt{T}^p\mathtt{AB}^q$ or $\mathtt{T}^p\mathtt{BA}^q$. Then $expo(a-b) \in \{\ell, \ell-1\}$.

Also, ℓ is the largest k such that

$$\sigma_{k+1}\sigma_k\sigma_{k-1} \in \quad \{\mathtt{TAA}, \mathtt{TAT}, \mathtt{ABA}, \mathtt{ABT}, \mathtt{BBA}, \mathtt{BBT}, \\ \mathtt{TBB}, \mathtt{TBT}, \mathtt{BAB}, \mathtt{BAT}, \mathtt{AAB}, \mathtt{AAT}\}.$$

But for each k, this condition is equivalent to

$$\lambda[k] = 1.$$

Thus, $expo(\lambda) = \ell$.

AMD RTL Language

- Based on Verilog, but intended specifically for RTL modeling of microprocessors
- Combinational assignments (wires):

```
sticksum[47:0] = esub ?
                   ina_add[47:0] ^ inb_add[47:0]
                 : ~(ina_add[47:0] ^ inb_add[47:0]);
```

- Sequential assignments (registers):

```
M4_sticky_co <= M3_sticky_co;
```

- Simple iteration construct:

```
found = 1'b0;
for (i=68; i>=0; i=i-1)
  if (lop[i] & ~found)
    begin
      found = 1'b1;
      lsa[6:0] = 7'h44 - i[6:0];
    end
```


Pipelines

A circuit is *combinational* if all of its signals are wires.

We shall say that a circuit is an *n-cycle simple pipeline* if there exists a mapping ϕ from its signals to $\{1, \ldots, n\}$ such that

(1) if s is an input then $\phi(s) = 1$;

(2) if s is a wire, then $\phi(s') = \phi(s)$ for each signal s' occurring in the expression assigned to s;

(3) if s is a register, then $\phi(s') = \phi(s) - 1$ for each signal s' occurring in the expression assigned to s;

(4) if s is an output, then $\phi(s) = n$.

If each sequential assignment of a simple pipeline is replaced by the corresponding combinational assignment, then the result is an "equivalent" combinational circuit.


```
module MULT;

//*******************************************************************
// Declarations
//*******************************************************************

//Precision and rounding control:
`define SNG   1'b0       // single
`define DBL   1'b1       // double
`define NRE   2'b00      // round to nearest
`define NEG   2'b01      // round to minus infinity
`define POS   2'b10      // round to plus infinity
`define CHP   2'b11      // truncate

//Parameters:

input x[79:0];              //first operand
input y[79:0];              //second operand
input rc[1:0];              //rounding control
input pc;                   //precision control
output z[79:0];             //product

//*******************************************************************
// First Cycle
//*******************************************************************

//Operand fields:
sgnx = x[79]; sgny = y[79];                       //signs
expx[14:0] = x[78:64]; expy[14:0] = y[78:64];     //exponents
sigx[63:0] <= x[63:0]; sigy[63:0] <= y[63:0];     //significands

//Sign of result:
sgnz <= sgnx ^ sgny;

//Biased exponent sum:
exp_sum[14:0] <= expx[14:0] + expy[14:0] + 15'h4001;

//Registers:
rc_C2[1:0] <= rc[1:0];  pc_C2 <= pc;
```

```
//******************************************************************
// Second Cycle
//******************************************************************

//Rounding constant -- overflow case

rconst_sing_of[127:0] =
  case(rc_C2[1:0])
    ‘NRE : {25’b1, 103’b0};
    ‘NEG  : sgnz ? {24’b0, {104 {1’b1}}} : 128’b0;
    ‘POS  : sgnz ? 128’b0 : {24’b0, {104 {1’b1}}};
    ‘CHP : 128’b0;
  endcase;

rconst_doub_of[127:0] =
  case(rc_C2[1:0])
    ‘NRE : {54’b1, 74’b0};
    ‘NEG  : sgnz ? {53’b0, {75 {1’b1}}} : 128’b0;
    ‘POS  : sgnz ? 128’b0 : {53’b0, {75 {1’b1}}};
    ‘CHP : 128’b0;
  endcase;

rconst_of[127:0] <= case(pc_C2)
                     ‘SNG : rconst_sing_of[127:0];
                     ‘DBL : rconst_doub_of[127:0];
                    endcase;

//Rounding constant -- no overflow:

rconst_nof[126:0] = rconst_of[127:1];

//Registers:

sgnz_C3 <= sgnz;
exp_sum_C3[14:0] <= exp_sum[14:0];
sigx_C3[63:0] <= sigx[63:0];  sigy_C3[63:0] <= sigy[63:0];
rc_C3[1:0] <= rc_C2[1:0];  pc_C3 <= pc_C2;
```



```
//******************************************************************
// Third Cycle
//******************************************************************

//Three sums are computed in parallel:

prod[127:0] = {64’b0, sigx_C3[63:0]} * {64’b0, sigy_C3[63:0]};

add_of[128:0] <= {1’b0, prod[127:0]} + {1’b0, rconst_of[127:0]};

add_nof[127:0] <= prod[127:0] + rconst_nof[127:0];


//overflow indicator:

overflow <= prod[127];


//Sticky bit:

sticky_of <= case(pc_C3)
               ‘SNG : |(prod[102:0]);
               ‘DBL : |(prod[73:0]);
             endcase;

sticky_nof <= case(pc_C3)
                ‘SNG : |(prod[101:0]);
                ‘DBL : |(prod[72:0]);
              endcase;

//Registers:

rc_C4[1:0] <= rc_C3[1:0];
pc_C4 <= pc_C3;
sgnz_C4 <= sgnz_C3;
exp_sum_C4[14:0] <= exp_sum_C3[14:0];
```



```
//******************************************************************
// Fourth Cycle
//******************************************************************

//Significand mask:
mask_of[127:0] =
  case (pc_C4)
    ‘SNG : (rc_C4[1:0] == ‘NRE) & ~sticky_of & ~add_of[103] ?
               {{23 {1’b1}}, 105’b0} : {{24 {1’b1}}, 104’b0};
    ‘DBL : (rc_C4[1:0] == ‘NRE) & ~sticky_of & ~add_of[74] ?
               {{52 {1’b1}}, 76’b0} : {{53 {1’b1}}, 75’b0};
  endcase;

mask_nof[126:0] =
  case (pc_C4)
    ‘SNG : (rc_C4[1:0] == ‘NRE) & ~sticky_nof & ~add_nof[102] ?
               {{23 {1’b1}}, 104’b0} : {{24 {1’b1}}, 103’b0};
    ‘DBL : (rc_C4[1:0] == ‘NRE) & ~sticky_nof & ~add_nof[73] ?
               {{52 {1’b1}}, 75’b0} : {{53 {1’b1}}, 74’b0};
  endcase;

//Carry bit:
carry_of = add_of[128];
carry_nof = add_nof[127];

//Significand and exponent:
sig_of[127:0] = {carry_of, 127’b0} |
                (add_of[127:0] & mask_of[127:0]);
sig_nof[126:0] = {carry_nof, 126’b0} |
                 (add_nof[126:0] & mask_nof[126:0]);
sigz[63:0] = overflow ? sig_of[127:64] : sig_nof[126:63];
exp_of[14:0] = exp_sum_C4[14:0] + {14’b0, carry_of} + 15’b1;
exp_nof[14:0] = exp_sum_C4[14:0] + {14’b0, carry_nof};
expz[14:0] = overflow ? exp_of[14:0] : exp_nof[14:0];

//Final result:
z[79:0] = {sgnz_C4, expz[14:0], sigz[63:0]};

endmodule
```


IEEE Compliance

According to IEEE Standard 754, each of the elementary floating point operations,

$$+,\ -,\ *,\ /,\ \text{and}\ \sqrt{\ },$$

> ... shall be performed as if it first produced an intermediate result correct to infinite precision and with unbounded range, and then rounded that result according to one of the [prescribed] modes....

Correctness of the Multiplier

Notation: Floating-point encodings are interpreted according to the extended precision format,

$$\mathcal{E} = (64, 15).$$

For a normal encoding a, we use the abbreviation

$$\hat{a} = decode(a, \mathcal{E}).$$

Theorem: Let x, y, rc, and pc be the inputs to the multiplier on cycle n, and let z be the output on cycle $n+3$. Assume that x and y are normal encodings, $0 \le rc < 4$, and $0 \le pc < 2$. Let

$$\mathcal{M} = \begin{cases} near & \text{if } rc = 0 \\ minf & \text{if } rc = 1 \\ inf & \text{if } rc = 2 \\ trunc & \text{if } rc = 3, \end{cases}$$

$$p = \begin{cases} 24 & \text{if } pc = 0 \\ 53 & \text{if } pc = 1, \end{cases}$$

and $\mathcal{I} = \mathcal{M}(\hat{x}\hat{y}, p)$.

If $\mathcal{I}$ is representable, then z is a normal encoding and

$$\hat{z} = \mathcal{I}.$$

Pipelines in ACL2

Each signal is represented by a function, automatically generated by an RTL-ACL2 translator:*

```
sig_nof[126:0] = {carry_nof, 126'b0} |
                 (add_nof[126:0] & mask_nof[126:0]);

sigz[63:0] = overflow ? sig_of[127:64] : sig_nof[126:63];
```

```
(DEFUN SIG_NOF (CARRY_NOF ADD_NOF MASK_NOF)
       (LOGIOR (CAT CARRY_NOF 0 126)
               (LOGAND ADD_NOF MASK_NOF)))

(DEFUN SIGZ (OVERFLOW SIG_OF SIG_NOF)
       (?? (OVERFLOW)
           (BITS SIG_OF 127 64)
           (BITS SIG_NOF 126 63)))
```

*Translator written by Art Flatau

Pipelines in ACL2 (cont.)

Iteration is translated into recursion:

```
found = 1'b0;
for (i=68; i>=0; i=i-1)
  if (lop[i] & ~found)
    begin
      found = 1'b1;
      lsa[6:0] = 7'h44 - i[6:0];
    end
```

```
(DEFUN LSA-AUX (LSA FOUND LOP I)
  (declare (xargs :measure (1+ i)))
  (IF (AND (INTEGERP I) (>= I 0))
      (IF (NOT (EQUAL (LOGAND (BITN LOP I)
                              (COMP1 FOUND 1))
                      0))
          (LSA-AUX (- 68 I) FOUND LOP (1- I))
        (LSA-AUX LSA 1 LOP (1- I)))
    LSA))

(DEFUN LSA (LOP)
  (LSA-AUX 0 0 LOP 68))
```


Pipelines in ACL2 (cont.)

Each output is represented as a function of inputs:

```
(DEFUN FMUL (X Y RC PC)
       (LET* ((SGNX (SGNX X))
              (SGNY (SGNY Y))
              (EXPX (EXPX X))
              (EXPY (EXPY Y))
              (SIGX (SIGX X))
              (SIGY (SIGY Y))
              (SGNZ (SGNZ SGNX SGNY))
              (EXP_SUM (EXP_SUM EXPX EXPY))
              (RC_C2 (RC_C2 RC))
              (PC_C2 (PC_C2 PC))
              (RCONST_SING_OF (RCONST_SING_OF SGNZ RC_C2))
              (RCONST_DOUB_OF (RCONST_DOUB_OF SGNZ RC_C2))
              (RCONST_OF (RCONST_OF RCONST_DOUB_OF PC_C2 RCONST_SING_OF))
              (RCONST_NOF (RCONST_NOF RCONST_OF))
              (SGNZ_C3 (SGNZ_C3 SGNZ))
              (EXP_SUM_C3 (EXP_SUM_C3 EXP_SUM))
              (SIGX_C3 (SIGX_C3 SIGX))
              (SIGY_C3 (SIGY_C3 SIGY))
                ...
              (EXP_SUM_C4 (EXP_SUM_C4 EXP_SUM_C3))
              (MASK_OF (MASK_OF PC_C4 RC_C4 STICKY_OF ADD_OF))
              (MASK_NOF (MASK_NOF PC_C4 RC_C4 STICKY_NOF ADD_NOF))
              (CARRY_OF (CARRY_OF ADD_OF))
              (CARRY_NOF (CARRY_NOF ADD_NOF))
              (SIG_OF (SIG_OF CARRY_OF ADD_OF MASK_OF))
              (SIG_NOF (SIG_NOF CARRY_NOF ADD_NOF MASK_NOF))
              (SIGZ (SIGZ OVERFLOW SIG_OF SIG_NOF))
              (EXP_OF (EXP_OF EXP_SUM_C4 CARRY_OF))
              (EXP_NOF (EXP_NOF EXP_SUM_C4 CARRY_NOF))
              (EXPZ (EXPZ OVERFLOW EXP_OF EXP_NOF))
              (Z (Z SGNZ_C4 EXPZ SIGZ)))
             Z))
```

Correctness of the Multiplier

```
;;Extended precision format, used internally by the multiplier:

(DEFUN EXTFMT () '(64 15))

;;The number represented by a bit vector in the extended format:

(DEFUN HAT (Z) (DECODE Z (EXTFMT)))

;;The rounding mode represented by the input parameter RC:

(DEFUN MODE (RC)
  (CASE RC (0 'NEAR) (1 'MINF)
           (2 'INF)  (3 'TRUNC)))

;;The precision represented by the input parameter PC:

(DEFUN PRECISION (PC)
  (CASE PC (0 24) (1 53)))

;;The statement of correctness:

(DEFTHM CORRECTNESS-OF-FMUL
    (LET ((IDEAL (RND (* (HAT X) (HAT Y))
                      (MODE RC)
                      (PRECISION PC)))
          (Z (FMUL X Y RC PC)))
      (IMPLIES (AND (NORMAL-ENCODING-P X (EXTFMT))
                    (NORMAL-ENCODING-P B (EXTFMT))
                    (MEMBER RC (LIST 0 1 2 3))
                    (MEMBER PC (LIST 0 1))
                    (REPP IDEAL (EXTFMT)))
               (AND (NORMAL-ENCODING-P Z (EXTFMT))
                    (= (HAT Z) IDEAL)))))
```


Proof Methodology

```
;;First, we introduce the following two definitions:

(DEFUN INPUT-SPEC (X Y RC PC)
  (AND (NORMAL-ENCODING-P X (EXTFMT))
       (NORMAL-ENCODING-P Y (EXTFMT))
       (MEMBER RC (LIST 0 1 2 3))
       (MEMBER PC (LIST 0 1))
       (REPP (RND (* (HAT X) (HAT Y)) (MODE RC) (PRECISION PC))
             (EXTFMT))))

(DEFUN OUTPUT-SPEC (X Y RC PC)
  (LET ((Z (FMUL X Y RC PC)))
    (AND (NORMAL-ENCODING-P Z (EXTFMT))
         (= (HAT Z)
            (RND (* (HAT X) (HAT Y))
                 (MODE RC)
                 (PRECISION PC))))))

(IN-THEORY (DISABLE INPUT-SPEC OUTPUT-SPEC))

;;The inputs (X*), (Y*), (RC*), and (PC*) are constrained to
;;satisfy the input specification:

(ENCAPSULATE ((X* () T) (Y* () T) (RC* () T) (PC* () T))
  (LOCAL (DEFUN X* () (ENCODE 1 (EXTFMT))))
  (LOCAL (DEFUN Y* () (ENCODE 1 (EXTFMT))))
  (LOCAL (DEFUN RC* () 0))
  (LOCAL (DEFUN PC* () 1))
  (DEFTHM INPUT-SPEC* (INPUT-SPEC (X*) (Y*) (RC*) (PC*)))
  (IN-THEORY (DISABLE INPUT-SPEC*)))
```



```
Next, we define a constant function corresponding to each signal:

(DEFUN SGNX* NIL (SGNX (X*)))
(DEFUN SGNY* NIL (SGNY (Y*)))
(DEFUN EXPX* NIL (EXPX (X*)))
(DEFUN EXPY* NIL (EXPY (Y*)))
(DEFUN SIGX* NIL (SIGX (X*)))
(DEFUN SIGY* NIL (SIGY (Y*)))
(DEFUN SGNZ* NIL (SGNZ (SGNX*) (SGNY*)))
(DEFUN EXP_SUM* NIL (EXP_SUM (EXPX*) (EXPY*)))
(DEFUN RC_C2* NIL (RC_C2 (RC*)))
(DEFUN PC_C2* NIL (PC_C2 (PC*)))
(DEFUN RCONST_SING_OF* NIL (RCONST_SING_OF (SGNZ*) (RC_C2*)))
(DEFUN RCONST_DOUB_OF* NIL (RCONST_DOUB_OF (SGNZ*) (RC_C2*)))
  ...
(DEFUN SIG_OF* NIL (SIG_OF (CARRY_OF*) (ADD_OF*) (MASK_OF*)))
(DEFUN SIG_NOF* NIL (SIG_NOF (CARRY_NOF*) (ADD_NOF*) (MASK_NOF*)))
(DEFUN SIGZ* NIL (SIGZ (OVERFLOW*) (SIG_OF*) (SIG_NOF*)))
(DEFUN EXP_OF* NIL (EXP_OF (EXP_SUM_C4*) (CARRY_OF*)))
(DEFUN EXP_NOF* NIL (EXP_NOF (EXP_SUM_C4*) (CARRY_NOF*)))
(DEFUN EXPZ* NIL (EXPZ (OVERFLOW*) (EXP_OF*) (EXP_NOF*)))
(DEFUN Z* NIL (Z (SGNZ_C4*) (EXPZ*) (SIGZ*)))

(DEFTHM FMUL-STAR-EQUIVALENCE
        (EQUAL (Z*)
               (FMUL (X*) (Y*) (RC*) (PC*)))
        :RULE-CLASSES NIL)

;;Now suppose that we are able to prove this theorem:

(DEFTHM Z*-SPEC
    (AND (NORMAL-ENCODING-P (Z*) (EXTFMT))
         (= (HAT (Z*))
            (RND (* (HAT (X*)) (HAT (Y*)))
                 (MODE (RC*))
                 (PRECISION (PC*)))))
  :RULE-CLASSES())
```



```
;;Then the desired correctness theorem can be derived as follows:

(DEFTHM OUTPUT-SPEC*
    (OUTPUT-SPEC (X*) (Y*) (RC*) (PC*))
  :HINTS (("GOAL" :IN-THEORY (ENABLE OUTPUT-SPEC)
                  :USE (Z*-SPEC FMUL-STAR-EQUIVALENCE))))

(DEFTHM FMUL-INPUT-OUTPUT
    (IMPLIES (INPUT-SPEC X Y RC PC)
             (OUTPUT-SPEC X Y RC PC))
  :HINTS (("GOAL"
           :IN-THEORY (ENABLE INPUT-SPEC*)
           :USE ((:FUNCTIONAL-INSTANCE OUTPUT-SPEC*
                   (X* (LAMBDA ()
                         (IF (INPUT-SPEC X Y RC PC) X (X*))))
                   (Y* (LAMBDA ()
                         (IF (INPUT-SPEC X Y RC PC) Y (Y*))))
                   (RC* (LAMBDA ()
                           (IF (INPUT-SPEC X Y RC PC) RC (RC*))))
                   (PC* (LAMBDA ()
                          (IF (INPUT-SPEC X Y RC PC) PC (PC*))))))))
  :RULE-CLASSES ())

(DEFTHM CORRECTNESS-OF-FMUL
    (LET ((IDEAL (RND (* (HAT X) (HAT Y))
                      (MODE RC)
                      (PRECISION PC)))
          (Z (FMUL X Y RC PC)))
      (IMPLIES (AND (NORMAL-ENCODING-P X (EXTFMT))
                    (NORMAL-ENCODING-P Y (EXTFMT))
                    (MEMBER RC (LIST 0 1 2 3))
                    (MEMBER PC (LIST 0 1))
                    (REPP IDEAL (EXTFMT)))
               (AND (NORMAL-ENCODING-P Z (EXTFMT))
                    (= (HAT Z) IDEAL))))
  :HINTS (("GOAL" :IN-THEORY (ENABLE INPUT-SPEC OUTPUT-SPEC)
                  :USE (FMUL-INPUT-OUTPUT))))
```