Chapter 10
Other Operations

This chapter is devoted to arithmetic functions and operations other than the four basic ones. The conversion of binary numbers to radix-B ones, and conversely, is dealt with in Sects. 10.1 and 10.2. An important particular case is $B = 10$ as human interfaces generally use decimal representations while internal computations are performed with binary circuits. In Sect. 10.3, several square rooting circuits are presented, based on digit-recurrence or convergence algorithms. Logarithms and exponentials are the topics of Sects. 10.4 and 10.5. Finally, the computation of trigonometric functions, based on the CORDIC algorithm [2, 3], is described in Sect. 10.6.

10.1 Binary to Radix-*B* Conversion (*B* even)

Assume that B is even and greater than 2. Consider the binary representation of x:

$$x = x_{n-1} \cdot 2^{n-1} + x_{n-2} \cdot 2^{n-2} + \cdots + x_1 \cdot 2 + x_0. \tag{10.1}$$

Taking into account that $B > 2$, the bits x_i can be considered as radix-B digits, and a simple conversion method consists of computing (10.1) in radix-B.

Algorithm 10.1: Binary to radix-B conversion

```
z := 0;
for i in 1 .. n loop
  z := z·2 + x_{n-i};
end loop;
x := z;
```

J.-P. Deschamps et al., *Guide to FPGA Implementation of Arithmetic Functions*, Lecture Notes in Electrical Engineering 149, DOI: 10.1007/978-94-007-2987-2_10,

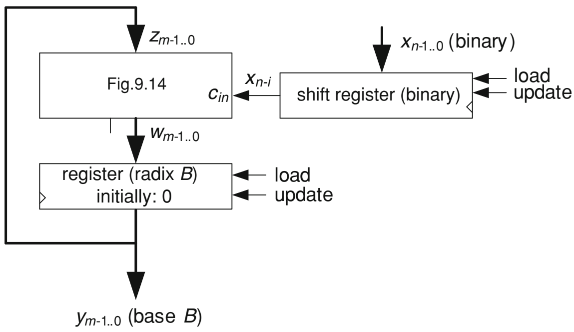

Fig. 10.1 Binary to radix-B converter

In order to compute $z{\cdot}2 + x_{n-i}$ the circuit of Fig. 9.14, with $c_{in} = x_{n-i}$ instead of 0, can be used. A sequential binary to radix-B converter is shown in Fig. 10.1. It is described by the following VHDL model.

```
main_component: doubling_circuit2 GENERIC MAP(n => m)
PORT MAP(x => z, z => w, c_in => xNminusI);
register_z: PROCESS(clk) ...
y <= z;
shift_register_x: PROCESS(clk) ...
xNminusI <= int_x(n-1);
```

The complete circuit also includes an n-state counter and a control unit. A complete model *BinaryToDecimal2.vhd* is available at the Authors' web page (B = 10).

The computation time of the circuit of Fig. 10.1 is equal to $n{\cdot}T_{clk}$ where T_{clk} must be greater than T_{LUT-k} (Sect. 9.3). Thus

$$T_{\text{binary-radix-}B} \cong n \cdot T_{LUT-k}. \tag{10.2}$$

10.2 Radix-*B* to Binary Conversion (*B* even)

Given a natural z, smaller than 2^n, its binary representation is deduced from the following set of integer divisions

$$z = q_1 \cdot 2 + r_0,$$
$$q_1 = q_2 \cdot 2 + r_1,$$
$$\ldots$$
$$q_{n-1} = q_n \cdot 2 + r_{n-1},$$

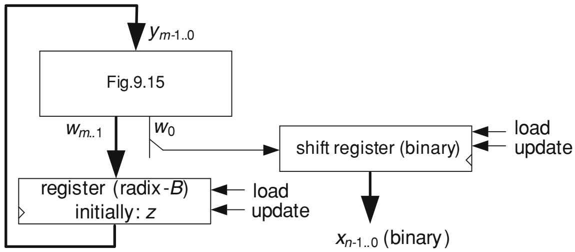

Fig. 10.2 Decimal to binary converter

so

$$z = q_n \cdot 2^n + r_{n-1} \cdot 2^{n-1} + r_{n-2} \cdot 2^{n-2} + \cdots + r_1 \cdot 2 + r_0.$$

As z is smaller than 2^n, then $q_n = 0$, and the binary representation of z is constituted by the set of remainders $r_{n-1}\, r_{n-2} \ldots r_1\, r_0$.

Algorithm 10.2: Radix-B to binary conversion

```
q0 := z;
for i in 0 .. n-1 loop
  ri := qi mod 2; qi+1 := ⌊qi/2⌋;
end loop;
x := rn-1 rn-2 ... r1 r0;
```

Observe that if $q_i = q_{i+1}{\cdot}2 + r_i$, then $q_i{\cdot}(B/2) = q_{i+1}{\cdot}B + r_i{\cdot}(B/2)$ where $r_i{\cdot}(B/2) < 2{\cdot}(B/2) = B$.

Algorithm 10.3: Radix-B to binary conversion, version 2

```
q0 := z;
for i in 0 .. n-1 loop
  yi := (B/2)·qi;
  qi+1 := ⌊yi/B⌋; ri := (yi mod B) mod 2;
end loop;
x := rn-1 rn-2 ... r1 r0;
```

In order to compute $(B/2){\cdot}q_i$ the circuit of Fig. 9.15 can be used. A sequential radix-B to binary converter is shown in Fig. 10.2. It is described by the following VHDL model.

```
main_component: multiply_by_five GENERIC MAP(n => m)
PORT MAP(x => q, z => w);
r <= w(0);
register_y: PROCESS(clk)
BEGIN
  IF clk'EVENT AND clk = '1' THEN
    IF load = '1' THEN q <= x;
    ELSIF update = '1' THEN q <= w(4*m+3 DOWNTO 4);
    END IF;
  END IF;
END PROCESS;
shift_register_z: PROCESS(clk)
BEGIN
  IF clk'EVENT AND clk = '1' THEN
    IF update = '1' THEN z <= r&z(n-1 DOWNTO 1);
    END IF;
  END IF;
END PROCESS;
```

The complete circuit also includes an n-state counter and a control unit. A complete model *DecimalToBinary2.vhd* is available at the Authors' web page (B = 10).

The computation time of the circuit of Fig. 10.2 is equal to $n \cdot T_{clk}$ where T_{clk} must be greater than T_{LUT-k} (Sect. 9.3). Thus

$$T_{\text{radix}-B-\text{binary}} \cong n \cdot T_{LUT-k}. \tag{10.3}$$

10.3 Square Rooters

Consider a $2n$-bit natural $X = x_{2n-1} \cdot 2^{2n-1} + x_{2n-2} \cdot 2^{2n-2} + \cdots + x_1 \cdot 2 + x_0$, and compute $Q = \lfloor X^{1/2} \rfloor$. Thus, $Q^2 \le X < (Q+1)^2$, and the difference $R = X - Q^2$ belongs to the range

$$0 \le R \le 2Q. \tag{10.4}$$

10.3.1 Restoring Algorithm

A digit recurrence algorithm consisting of n steps is defined. At each step two numbers are generated:

$$Q_i = q_{n-1} \cdot 2^{i-1} + q_{n-2} \cdot 2^{i-2} + \cdots + q_{n-i+1} \cdot 2 + q_{n-i} \text{ and } R_i = X - \left(Q_i \cdot 2^{n-i}\right)^2,$$

such that

$$0 \le R_i < (1 + 2Q_i)2^{2(n-i)}. \tag{10.5}$$

After n steps, $0 \le R_n < 1 + 2Q_n$, that is (10.4) with $Q = Q_n$ and $R = R_n$.

Initially define $Q_0 = 0$ and $R_0 = X$, so condition (10.5) amounts to $0 \le X < 2^{2n}$. Then, at step i, compute Q_i and R_i in function of Q_{i-1} and R_{i-1}:

$$Q_i = 2Q_{i-1} + q_{n-i} \text{ where } q_{n-i} \in \{0, 1\},$$
$$R_i = X - \left(Q_i \cdot 2^{n-i}\right)^2 = X - \left((2Q_{i-1} + q_{n-i})2^{n-i}\right)^2$$
$$= X - \left(Q_{i-1} \cdot 2^{n-i+1}\right)^2 - (q_{n-i} + 4Q_{i-1})q_{n-i} \cdot 2^{2(n-i)} = R_{i-1} - (q_{n-i} + 4Q_{i-1})q_{n-i}2^{2(n-i)}.$$

The value of q_{n-i} is chosen in such a way that condition (10.5) holds. Consider two cases:

- If $R_{i-1} < (1 + 4Q_{i-1})2^{2(n-i)}$, then $q_{n-i} = 0$, $Q_i = 2Q_{i-1}, R_i = R_{i-1}$.
  As $R_i = R_{i-1} < (1 + 4Q_{i-1})2^{2(n-i)} = (1 + 2Q_i)2^{2(n-i)}$ and $R_i = R_{i-1} \ge 0$, condition (10.5) holds.
- If $R_{i-1} \ge (1 + 4Q_{i-1})2^{2(n-i)}$, then $q_{n-i} = 1$, $Q_i = 2Q_{i-1} + 1$, $R_i = R_{i-1} - (1 + 4Q_{i-1})2^{2(n-i)}$, so that $R_i \ge 0$ and $R_i < (1 + 2Q_{i-1})2^{2(n-i+1)} - (1 + 4Q_{i-1})2^{2(n-i)} = (3 + 4Q_{i-1})2^{2(n-i)} = (1 + 2Q_i)2^{2(n-i)}$.

Algorithm 10.4: Square root, restoring algorithm

```
Q0 := 0; R0 := X;
for i in 1 to n loop
  Pi-1 := (1 + 4·Qi-1)·2^2(n-i);
  if Pi-1 ≤ Ri-1 then qn-i := 1; Ri := Ri-1 - Pi-1;
  else qn-i := 0; Ri := Ri-1;
  end if;
  Qi := 2·Qi-1 + qn-i;
end loop;
Q := Qn;
```

Q_i is an i-bit number, $R_i < (1 + 2Q_i)2^{2(n-i)} = Q_i \,\&\, 1 \& 0 0 \cdots 0$ is a $(2n{-}i{+}1)$-bit number, and $P_{i-1} = (1 + 4Q_{i-1})2^{2(n-i)} = Q_i \,\&\, 01 \& 00 \cdots 0$ a $(2n + 2 - i)$-bit number.

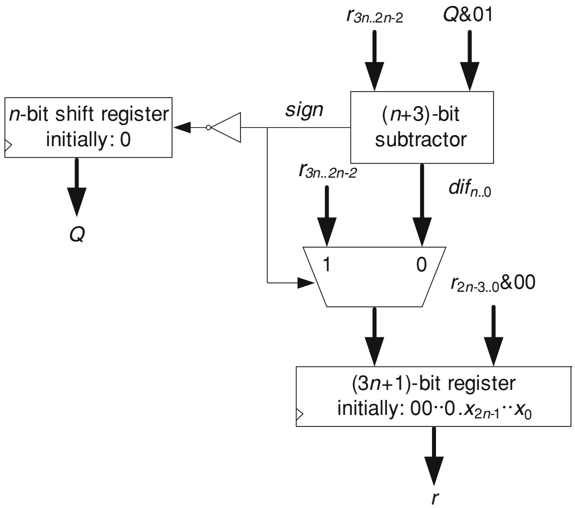

Fig. 10.3 Square root computation: data path

An equivalent algorithm is obtained if P_i and R_i are replaced by $p_{i-1} = P_{i-1}/2^{2(n-i)}$ and $r_i = R_i/2^{2(n-i)}$.

Algorithm 10.5: Square root, restoring algorithm, version 2

```
Q0 := 0; r0 := X/2^(2n);
for i in 1 to n loop
  p(i-1) := 1 + 4Q(i-1);
  if p(i-1) ≤ 4r(i-1) then
    q(n-i) := 1; r(i) := 4r(i-1) - p(i-1);
  else q(n-i) := 0; r(i) := 4r(i-1);
  end if;
  Q(i) := 2Q(i-1) + q(n-i);
end loop;
Q := Q(n); R := r(n);
```

As before, Q_i is an i-bit number, r_i is a (2n−i+1)-bit fixed-point number with 2(n−i) fractional bits and an (i+1)-bit integer part, and p_{i-1} a (2n−i+2)-bit fixed-point number with 2(n−i) fractional bits and an (i+2)-bit integer part.

A sequential implementation is shown in Fig. 10.3. It can be described by the following VHDL model.

```
dif <= r(3*n DOWNTO 2*n-2) - ('0'&q&"01");
WITH dif(n+2) SELECT next_r <=
  r(3*n-2 DOWNTO 0)&"00" WHEN '1',
  dif(n DOWNTO 0)&r(2*n-3 DOWNTO 0) &"00" WHEN OTHERS;
remainder_register: PROCESS(clk)
BEGIN
  IF clk'EVENT AND clk = '1' THEN
    IF load = '1' THEN r(2*n-1 DOWNTO 0) <= x;
      r(3*n DOWNTO 2*n) <= (OTHERS => '0');
    ELSIF update = '1' THEN r <= next_r;
    END IF;
  END IF;
END PROCESS;
remainder <= r(3*n DOWNTO 2*n);
quotient_register: PROCESS(clk)
BEGIN
  IF clk'EVENT AND clk = '1' THEN
    IF load = '1' THEN q <= (OTHERS => '0');
    ELSIF update = '1' THEN
      q <= q(n-2 DOWNTO 0)&NOT(dif(n+2));
    END IF;
  END IF;
END PROCESS;
root <= q;
```

The only computational resource is an (n+3)-bit subtractor so that the computation time is approximately equal to $n \cdot T_{adder}(n)$. The complete circuit includes an n-bit counter and a control unit. A generic VHDL model *SquareRoot.vhd* is available at the Authors' web page.

Another equivalent algorithm is obtained if P_i, Q_i and R_i are replaced by $p_i = P_i/2^{2n-i-1}, q_i = Q_i/2^i, r_i = R_i/2^{2n-i}$.

Algorithm 10.6: Square root, restoring algorithm, version 3

```
q0 := 0; r0 := 0.x2n-1x2n-2...x0;
for i in 1 to n loop
  pi-1 := 2qi-1 + 2^-i;
  if pi-1 ≤ 2ri-1 then qi := qi-1+ 2^-i; ri := 2ri-1 - pi-1;
  else ri := 2ri-1;
  end if;
end loop;
Q := qn2^n; R := rn2^n;
```

Algorithm 10.6 is similar to the restoring algorithm defined in Chap. 21 of [1]. Its implementation is left as an exercise.

10.3.2 Non-Restoring Algorithm

Instead of computing R_i and Q_i as in Algorithm 10.4, an alternative option is the following. Define $R_i = R_{i-1} - (1 + 4Q_{i-1})2^{2(n-i)}$, whatever the sign of R_i. If R_i is non-negative, then its value is the same as before. If R_i is negative then it is equal to $R_{i\,restoring} - (1 + 4Q_{i-1})2^{2(n-i)}$ where $R_{i\ restoring}$ is the value that would have been computed with Algorithm 10.4. Then, at the next step, Q_i and R_{i+1} are computed as follows:

- if R_i is non-negative, then $Q_i = 2Q_{i-1} + 1$ and $R_{i+1} = R_i - (1 + 4Q_i)\,2^{2(n-i-1)}$,
- if R_i is negative, then $Q_i = 2Q_{i-1}$ and $R_{i+1} = R_{i\,restorig} - (1 + 4Q_i)2^{2(n-i-1)} = R_i + (1 + 4Q_{i-1})2^{2(n-i)} - (1 + 4Q_i)2^{2(n-i-1)} = R_i + (1 + 2Q_i)2^{2(n-i)} -(1 + 4Q_i)2^{2(n-i-1)} = R_i + (3 + 4Q_i)2^{2(n-i-1)}$.

Algorithm 10.7: Square root, non-restoring algorithm

```
Q0 := 0; R0 := X;
R1 := R0 - 2^(2(n-1));
for i in 1 to n loop
  if Ri ≥ 0 then Qi := 2·Qi-1 + 1; Ri+1 = Ri-(1 + 4·Qi)·2^(2(n-i-1));
  else Qi := 2·Qi-1; Ri+1 = Ri + (3 + 4·Qi)·2^(2(n-i-1));
  end if;
end loop;
Q := Qn;
```

Q_i is an i-bit number and R_i is an (i+2)-bit signed number.

An equivalent algorithm is obtained if P_i and R_i are replaced by $p_{i-1} = P_{i-1}/2^{2(n-i)}, r_i = R_i/2^{2(n-i)}$.

Algorithm 10.8: Square root, non-restoring algorithm, version 2

```
Q0 := 0; r0 := 0.x2n-1 x2n-2 ... x0;
r1 := 4·r0 - 1;
for i in 1 to n loop
  if ri ≥ 0 then Qi := 2·Qi-1 + 1; ri+1 = 4·ri - (1 + 4·Qi);
  else Qi := 2·Qi-1; ri+1 = 4·ri + (3 + 4·Qi);
  end if;
end loop;
Q := Qn;
```

As before, Q_i is an i-bit number and r_i a (2n+1)-bit fixed-point number $a_{2n} \cdot a_{2n-1}\, a_{2n-2} \ldots a_0$ initially equal to $0.x_{2n-1}\, x_{2n-2}\, \ldots x_0$.

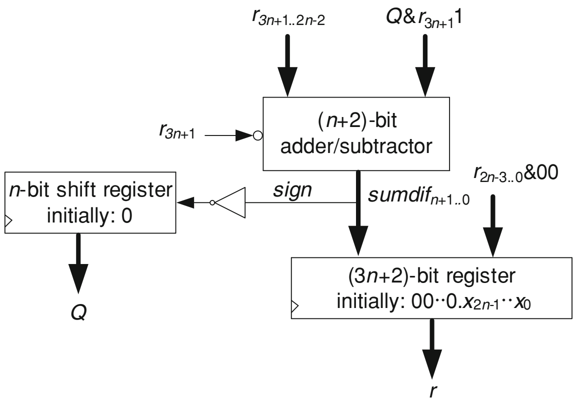

Fig. 10.4 Square root computation: non-restoring algorithm

In this case $r_n 2^n$ is the remainder only if r_n is non-negative. In fact, the remainder is equal to $(r_{n-i} \cdot 4^i) \cdot 2^n$ where r_{n-i} is the last non-negative remainder.

A sequential implementation is shown in Fig. 10.4. It can be described by the following VHDL model.

```
left_operand <= r(3*n-1 DOWNTO 2*n-2);
right_operand <= q&r(3*n+1)&'1';
WITH r(3*n+1) SELECT sumdif <=
  left_operand - right_operand WHEN '0',
  left_operand + right_operand WHEN OTHERS;
next_r <= sumdif&r(2*n-3 DOWNTO 0) &"00";
remainder_register: PROCESS(clk) ...
remainder <= r(3*n DOWNTO 2*n);
quotient_register: PROCESS(clk)
BEGIN
  IF clk'EVENT AND clk = '1' THEN
    IF load = '1' THEN q <= (OTHERS => '0');
    ELSIF update = '1' THEN
      q <= q(n-2 DOWNTO 0)&NOT(sumdif(n+1));
    END IF;
  END IF;
END PROCESS;
root <= q;
```

The only computation resource is an $(n+2)$-bit adder/subtractor so that the computation time is again approximately equal to $n \cdot T_{adder}(n)$. The complete circuit includes an n-bit counter and a control unit. A generic VHDL model *SquareRoot3.vhd* is available at the Authors' web page.

10.3.3 Fractional Numbers

Assume that X is a $2(n+p)$-bit fractional number $x_{2n-1}\,x_{2n-2}\cdots x_1\,x_0.\,x_{-1}\,x_{-2}\cdots x_{-2p}$. The square root Q of X, with an accuracy of p fractional bits, is defined as follows:

$$Q = q_{n-1}2^{n-1} + q_{n-2}2^{2n-2} + \ldots + q_0 + q_{-1}2^{-1} + q_{-2}2^{-2} + \ldots + q_{-p}2^{-p},$$
$$Q^2 \le X \text{ and } (Q + 2^{-p})^2 > X,$$

so that the remainder $R = X - Q^2$ belongs to the range $0 \le R < 2^{1-p}Q + 2^{-2p}$, that is to say

$$0 \le R \le Q \cdot 2^{1-p}.$$

In order to compute Q, first substitute X by $X' = X{\cdot}2^{2p}$, which is a natural, and then compute the square root $Q' = q_{n+p-1}\,q_{n+p-2}\ldots q_1\,q_0$ of X', and the corresponding remainder $R' = r_{n+p}\,r_{n+p-1}\ldots r_1\,r_0$, using for that one of the previously defined algorithms. Thus, $X' = (Q')^2 + R'$, with $0 \le R' \le 2Q'$, so that $X = (Q' \cdot 2^{-p})^2 + R' \cdot 2^{-2p}$, with $0 \le R'{\cdot}2^{-2p} \le 2Q'{\cdot}2^{-2p}$. Finally, define $Q = Q'{\cdot}2^{-p}$ and $R = R'{\cdot}2^{-2p}$. Thus

$$X = Q^2 + R, \text{ with } 0 \le R \le Q{\cdot}2^{1-p},$$

where $Q = q_{n+p-1}\,q_{n+p-2}\ldots q_p \cdot q_{p-1}\ldots q_1\,q_0$ and $R = r_{n+p}\,r_{n+p-1}\ldots r_{2p} \cdot r_{2p-1}\ldots r_1\,r_0$ is smaller than or equal to $Q{\cdot}2^{1-p}$.

Comment 10.1
The previous method can also be used for computing the square root of a natural $X = x_{2n-1}\,x_{2n-2}\ldots x_1\,x_0$ with an accuracy of p bits: represent X as an $(n+p)$-bit fractional number $x_{2n-1}\,x_{2n-2}\ldots x_1\,x_0 \cdot 00\ldots 0$ and use the preceding method.

10.3.4 Convergence Methods (Newton–Raphson)

Instead of a digit-recurrence algorithm, an alternative option is the Newton–Raphson convergence method. The following iteration can be used for computing $X^{1/2}$

$$x_{i+1} = (1/2) \cdot (x_i + X/x_i).$$

It corresponds to the graphical construction of Fig. 10.5.

First check that $(1/2){\cdot}(x + X/x)$ is a function whose minimum value, within the half plane $x > 0$, is equal to $X^{1/2}$, and is obtained when $x_i = X^{1/2}$. Thus, whatever the initial value x_0, x_i is greater than or equal to $X^{1/2}$ for all $i > 0$. Furthermore, if $x_i > X^{1/2}$ then $x_{i+1} < (1/2) \cdot \left(x_i + X/X^{1/2}\right) = (1/2) \cdot \left(x_i + X^{1/2}\right) < (1/2) \cdot (x_i + x_i) = x_i$. Thus, either $X^{1/2} < x_{i+1} < x_i$ or $X^{1/2} = x_{i+1} = x_i$. For x_0 choose a first rough approximation of $X^{1/2}$. As regards the computation of X/x_i, observe that if $x_i \ge X^{1/2}$ and $X < 2^{2n}$, then $x_i \cdot 2^n > X^{1/2} \cdot X^{1/2} = X$. So, compute $q \cong X/(x_i{\cdot}2^n)$, with an

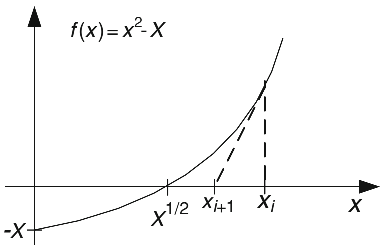

Fig. 10.5 Newton–Raphson method: computation of $X^{1/2}$

accuracy of $p+n$ fractional bits, using any division algorithm, so that $X \cdot 2^{n+p} = q \cdot x_i \cdot 2^n + r$, with $r < x_i \cdot 2^n$, and $X = Q \cdot x_i + R$, where $Q = q \cdot 2^{-p}$ and $R = (r/x_i) \cdot 2^{-(n+p)} < 2^{-p}$.

An example of implementation *SquareRootNR4.vhd* is available at the Authors' web page. The corresponding data path is shown in Fig. 10.6. The initial value x_0 must be defined in such a way that $x_0 \cdot 2^n > X$. In the preceding example, *initial_y* = x_0 is defined so that *initial_y*$(n+p \cdot\cdot\, n+p-4) \cdot 2^{-2}$ is an approximation of the square root of $X_{2n-1\,\cdot\cdot\,2n-4}$ and that *initial_y*$\cdot 2^n$ is greater than X.

```
first_bits <= x(2*n-1 DOWNTO 2*n-4);
initial_y(n+p DOWNTO n+p-4) <=
  table_x0(CONV_INTEGER(first_bits));
```

table_x0 is a constant array defined within a user package:

```
TYPE table IS ARRAY(0 TO 15) OF STD_LOGIC_VECTOR(4 DOWNTO 0);
CONSTANT table_x0: table := (
"00001",
"00101",
"00110",
"00111",
"01001",
"01001",
"01010",
"01011",
"01100",
"01101",
"01101",
"01110",
"01110",
"01111",
"01111",
"10000");
```

The end of computation is detected when $x_{i+1} = x_i$.

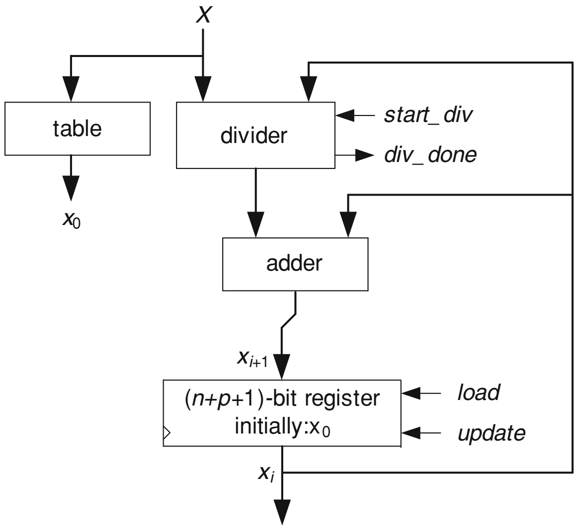

Fig. 10.6 Newton–Raphson method: data path

Comment 10.2
Every iteration step includes a division, an operation whose complexity is similar to that of a complete square root computation using a digit recurrence algorithm. Thus, this type of circuit is generally not time effective.

Another method is to first compute $X^{-1/2}$. A final multiplication computes $X^{1/2} = X^{-1/2} \cdot X$. The following iteration can be used for computing $X^{-1/2}$

$$x_{i+1} = (x_i/2) \cdot \left(3 - x_i^2 \cdot X\right),$$

where the initial value x_0 belongs to the range $0 < x_0 \leq X^{-1/2}$. The corresponding graphical construction is shown in Fig. 10.7.

The corresponding circuit does not include dividers, only multipliers and an adder. The implementation of this second convergence algorithm is left as an exercise.

10.4 Logarithm

Given an n-bit normalized fractional number $x = 1{\cdot}x_{-1}\ x_{-2} \cdots x_{-n}$, compute $y = log_2 x$ with an accuracy of p fractional bits. As x belongs to the interval $1 \leq x < 2$, its base-2 logarithm is a non-negative number smaller than 1, so $y = 0.y_{-1}\ y_{-2} \cdots y_{-p}$.

If $y = log_2 x$, then $x = 2^{0 \cdot y_{-1}\, y_{-2} \ldots y_{-p} \cdots}$, so that $x^2 = 2^{y_{-1} \cdot y_{-2} \cdots y_{-p} \cdots}$. Thus

- if $x^2 \geq 2 : y_{-1} = 1$ and $x^2/2 = 2^{0.y_{-2} \cdots y_{-p} \cdots}$;
- if $x^2 < 2 : y_{-1} = 0$ and $x^2 = 2^{0.y_{-2} \cdots y_{-p} \cdots}$.

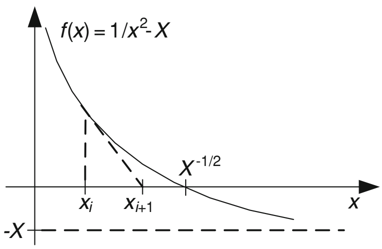

Fig. 10.7 Newton–Raphson method: computation of $X^{-1/2}$

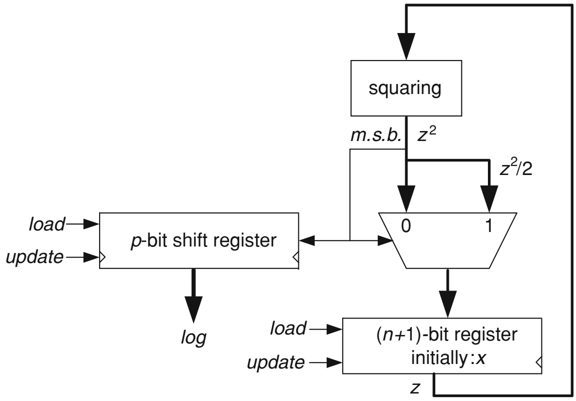

Fig. 10.8 Logarithm: data path

The following algorithm computes y:

Algorithm 10.9: Base-2 logarithm

```
z = x;
for i in 1 to p loop
  z := z²;
  if z ≥ 2 then y₋ᵢ := 1; z := z/2;
  else y₋ᵢ := 0;
end loop;
```

The preceding algorithm can be executed by the data path of Fig. 10.8 to which corresponds the following VHDL model.

```
square <= z*z;
WITH square(2*n+1) SELECT next_z <=
  square(2*n+1 DOWNTO n+1) WHEN '1',
  square(2*n DOWNTO n) WHEN OTHERS;
register_z: PROCESS(clk) ...
shift_register: PROCESS(clk) ...
```

A complete VHDL model *Logarithm.vhd* is available at the Authors' web page.

Comments 10.3

1. If x belongs to the interval $2^{n-1} \leq x < 2^n$, then it can be expressed under the form $x = 2^{n-1} \cdot y$ where $1 \leq y < 2$, so that $log_2 x = n - 1 + log_2 y$.
2. If the logarithm in another base, say b, must be computed, then the following relation can be used: $log_b x = log_2 x / log_2 b$.

10.5 Exponential

Given an n-bit fractional number $x = 0.x_{-1}\, x_{-2} \ldots x_{-n}$, compute $y = 2^x$ with an accuracy of p fractional bits. As $x = x_{-1}2^{-1} + x_{-2}2^{-2} + \ldots + x_{-n}2^{-n}$, then

$$2^x = \left(2^{2^{-1}}\right)^{x_{-1}} \left(2^{2^{-2}}\right)^{x_{-2}} \ldots \left(2^{2^{-n}}\right)^{x_{-n}}.$$

If all the constant values $a_i = 2^{2^{-i}}$ are computed in advance, then the following algorithm computes 2^x.

Algorithm 10.10: Exponential 2^x

```
z := 1;
for i in 1 to n loop
  if x-i = 1 then z := z·ai; end if;
end loop;
```

The preceding algorithm can be executed by the data path of Fig. 10.9.

The problem is accuracy. Assume that all a_i's are computed with m fractional bits so that the actual operand a_i' is equal to $a_i - \varepsilon_i$, where ε_i belongs to the range

$$0 \leq \varepsilon_i < 2^{-m}. \tag{10.6}$$

Consider the worst case, that is $y = 2^{0.11 \cdots 1}$. Then the obtained value is $y' = (a_1 - \varepsilon_1)(a_2 - \varepsilon_2) \ldots (a_n - \varepsilon_n)$. If second or higher order products $\varepsilon_i\, \varepsilon_j \cdots \varepsilon_k$ are not taken into account, then $y' \cong y - (\varepsilon_1\, a_2 \cdots a_n + a_1\, \varepsilon_2 \cdots a_n + a_1 \cdots a_{n-1}\, \varepsilon_n)$. As all products $p_1 = a_2 \cdots a_n$, $p_2 = a_1\, a_3 \cdots a_n$, etc., belong to the range $1 < p_i < 2$, and ε_i to (10.6), then

$$y - y' < 2 \cdot n \cdot 2^{-m-}. \tag{10.7}$$

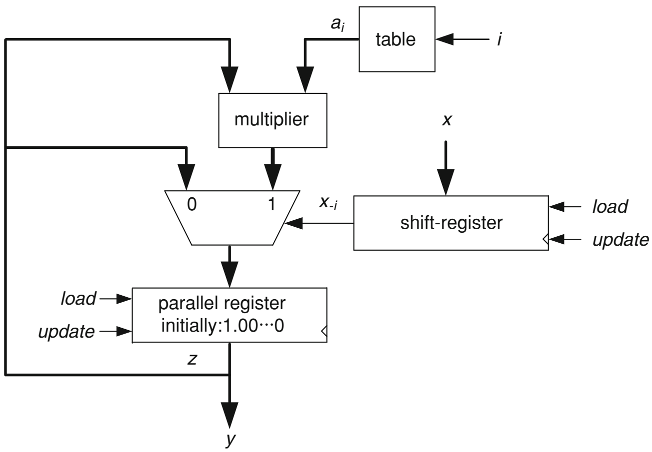

Fig. 10.9 Exponential: data path

Relation (10.7) would define the maximum error if all products were computed exactly, but it is not the case. At each step the obtained product is rounded. Thus Algorithm 10.8 successively computes

$$z_2 > a_1' \cdot a_2' - 2^m,$$
$$z_3 > \left(a_1' \cdot a_2' - 2^m\right) \cdot a_3' - 2^m = a_1' \cdot a_2' \cdot a_3' - 2^{-m}\left(1 + a_3'\right),$$
$$z_4 > \left(a_1' \cdot a_2' \cdot a_3' - 2^{-m}\left(1 + a_3'\right)\right) \cdot a_4' - 2^m = a_1' \cdot a_2' \cdot a_3' \cdot a_4' - 2^{-m}\left(1 + a_4' + a_3' \cdot a_4'\right),$$

and so on. Finally

$$\begin{aligned} z_n &> y' - 2^{-m}\left(1 + a_n' + a_{n-1}' \cdot a_n' + \cdots + a_3' \cdot a_4' \cdots a_n'\right) \\ &> y' - 2^{-m}(1 + 2(n-2)) > y' - 2 \cdot n \cdot 2^{-m}. \end{aligned} \tag{10.8}$$

Thus, from (10.7) and (10.8), the maximum error $y - z_n$ is smaller than $4{\cdot}n{\cdot}2^{-m}$. In order to obtain the result y with p fractional bits, the following relation must hold true: $4{\cdot}n{\cdot}2^{-m-} \leq 2^{-p}$, and thus

$$m \geq p + log_2 n + 2. \tag{10.9}$$

As an example, with $n = 8$ and $p = 12$, the internal data must be computed with $m = 17$ fractional bits.

The following VHDL model describes the circuit of Fig. 10.9.

```
a <= powers(count)(23 DOWNTO 24 - m);
product <= ('1'&a) * z;
WITH int_x(n-1) SELECT next_z <=
  product(2*m DOWNTO m) WHEN '1', z WHEN OTHERS;
register_z: PROCESS(clk) ...
y <= z(m DOWNTO m-p);
shift_register_x: PROCESS(clk) ...
```

powers is a constant array defined within a user package; it stores the fractional part of a_i with 24 bits:

```
TYPE table IS ARRAY(0 TO 7) OF STD_LOGIC_VECTOR(23 DOWNTO 0);
CONSTANT powers: table := (
 x"6a09e6",
 x"306fed",
 x"172b83",
 x"0b5586",
 x"059b0d",
 x"02c9a3",
 x"0163da",
 x"00b1af");
```

A complete VHDL model *Exponential.vhd* is available at the Authors' web page.

Instead of storing the constants a_i, an alternative solution is to store a_n, and to compute the other values on the fly:

$$a_{i-1} = 2^{2^{-i+1}} = \left(2^{2^{-i}}\right)^2 = a_i^2.$$

Algorithm 10.11: Exponential 2 ^{x}, version 2

```
z := 1; a := a_n;
for i in 0 to n-1 loop
  if x_{n-i} = 1 then z := z·a; end if;
  a := a·a;
end loop;
```

The preceding algorithm can be executed by the data path of Fig. 10.9 in which the table is substituted by the circuit of Fig. 10.10.

Once again, the problem is accuracy. In this case there is an additional problem: in order to get all coefficients a_i with an accuracy of m fractional bits, they must be computed with an accuracy of $k > m$ bits. Algorithm 10.11 successively computes

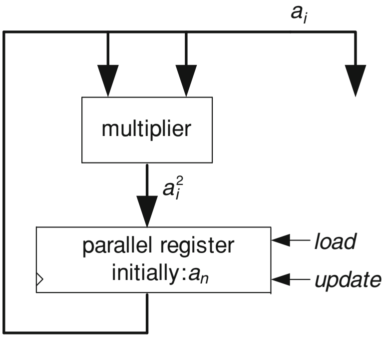

Fig. 10.10 Computation of a_i on the fly

$a_n^{'} > a_n - 2^{-k}, a_{n-1}^{'} > \left(a_n - 2^{-k}\right)^2 - 2^{-k} \cong a_n^2 - 2a_n 2^{-k} - 2^{-k} = a_{n-1} - 2^{-k}$
$(1 + 2a_n), a_{n-2}^{'} > \left(a_{n-1} - 2^{-k}(1 + 2a_n)\right)^2 - 2^{-k} \cong a_{n-1}^2 - 2a_{n-1}\, 2^{-k}(1 + 2a_n)$
$-2^{-k} = a_{n-2} - 2^{-k}(1 + 2a_{n-1} + 4a_{n-1}\, a_n), a_{n-3}^{'} > \left(a_{n-2} - 2^{-k}\right.$
$\left.(1 + 2a_{n-1} + 4a_{n-1}a_n)\right)^2 \quad -2^{-k} \cong a_{n-2}^2 - 2a_{n-2}2^{-k}(1 + 2a_{n-1} + 4a_{n-1}a_n) - 2^{-k}$
$= a_{n-3} \ -2^{-k}(1 + 2a_{n-2} + 4a_{n-2}\, a_{n-1} + 8a_{n-2}\, a_{n-1}\, a_n),$

and so on. Finally

$$\begin{aligned} a_1^{'} &> a_1 - 2^{-k}\left(1 + 2a_2 + 4a_2a_3 + \cdots + 2^{n-2}a_2\, a_3 \ldots a_n\right) \\ &> a_1 - 2^{-k}\left(1 + 4 + 8 + \cdots + 2^{n-1}\right) \\ &= a_1 - 2^{-k}(2^n - 3). \end{aligned}$$

In conclusion, $a_1 - a_1' < 2^{-k}(2^n - 3) < 2^{n-k}$. The maximum error is smaller than 2^{-m} if $n-k \leq -m$, that is $k \geq n + m$. Thus, according to (10.9)

$$k \geq n + p + log_2 n + 2.$$

As an example, with $n = 8$ and $p = 8$, the coefficients a_i (Fig. 10.10) are computed with $k = 21$ fractional bits and z (Fig. 10.9) with 13 fractional bits.

A complete VHDL model *Exponential2.vhd*, in which a_n, expressed with k fractional bits, is a generic parameter, is available at the Authors' web page.

Comment 10.3

Given an n-bit fractional number x and a number $b > 2$, the computation of $y = b^x$, with an accuracy of p fractional bits, can be performed with Algorithm 10.10 if the constants a_i are defined as follows:

$$a_i = b^{2^{-i}}.$$

So, the circuit is the same, but for the definition of the table which stores the constants a_i. In particular, it can be used for computing e^x or 10^x.

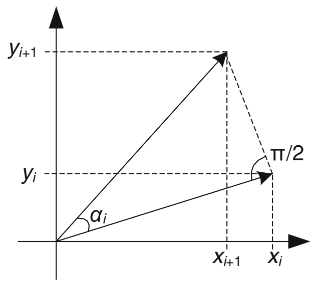

Fig. 10.11 Pseudo-rotation

10.6 Trigonometric Functions

A digit-recurrence algorithm for computing $e^{jz} = \cos z + j{\cdot}\sin z$, with $0 \le z \le \pi/2$, similar to Algorithm 10.10 can be defined. In the modified version, the operations are performed over the complex field.

Algorithm 10.12: Exponential $e^{jz}, z = z_0 \cdot z_{-1} z_{-2} \ldots z_{-n}$

```
(eR, eI) := (1, 0);
for i in 0 to n loop
  if z-i = 1 then (eR, eI) := (eR·aRi-eI·aIi, eR·aIi+eI·aRi);
  end if;
end loop;
```

The constants a_{Ri} and a_{Ii} are equal to cos 2^{-i} and sin 2^{-i}, respectively. The synthesis of the corresponding circuit is left as an exercise.

A more efficient algorithm, which does not include multiplications, is CORDIC [2, 3]. It is a convergence method based on the graphical construction of Fig. 10.11. Given a vector (x_i, y_i), then a pseudo-rotation by α_i radians defines a rotated vector (x_{i+1}, y_{i+1}) where

$$x_{i+1} = x_i - y_i \cdot \tan\alpha_i = (x_i \cdot \cos\alpha_i - y_i \cdot \sin\alpha_i) \cdot (1 + \tan^2\alpha_i)^{0.5},$$
$$y_{i+1} = y_i + x_i \cdot \tan\alpha_i = (y_i \cdot \cos\alpha_i + x_i \cdot \sin\alpha_i) \cdot (1 + \tan^2\alpha_i)^{0.5}.$$

In the previous relations, $x_i{\cdot}\cos\alpha_i - y_i{\cdot}\sin\alpha_i$ and $y_i{\cdot}\cos\alpha_i + x_i{\cdot}\sin\alpha_i$ define the vector obtained after a (true) rotation by α_i radians. Therefore, if an initial vector (x_0, y_0) is rotated by successive angles $\alpha_0, \alpha_1, \cdots, \alpha_{n-1}$, then the final vector is (x_n, y_n) where

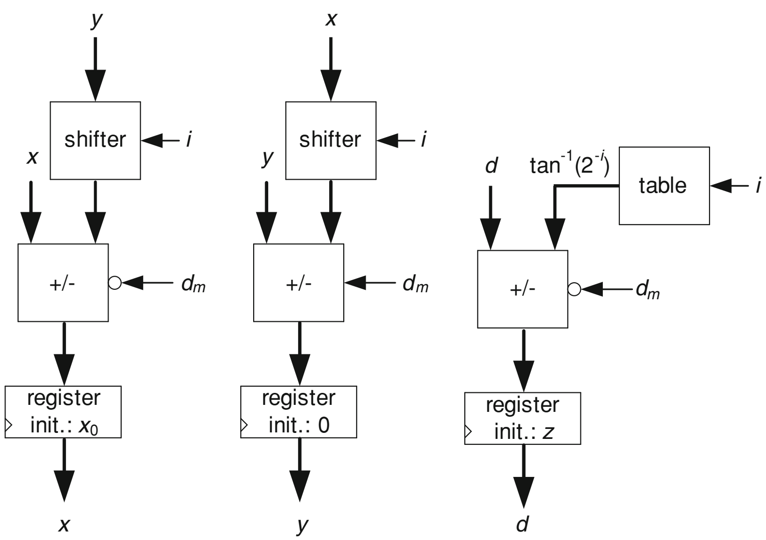

Fig. 10.12 Data path executing CORDIC

$$x_n = (x_0 \cdot \cos\alpha - y_0 \cdot \sin\alpha)(1 + \tan^2\alpha_0)^{0.5}(1 + \tan^2\alpha_1)^{0.5} \ldots (1 + \tan^2\alpha_{n-1})^{0.5},$$
$$y_n = (y_0 \cdot \cos\alpha + x_0 \cdot \sin\alpha)(1 + \tan^2\alpha_0)^{0.5}(1 + \tan^2\alpha_1)^{0.5} \ldots (1 + \tan^2\alpha_{n-1})^{0.5},$$

with $\alpha = \alpha_0 + \alpha_1 + \cdots + \alpha_{n-1}$. The CORDIC method consists in choosing angles α_i such that

- $x_i - y_i \cdot \tan\alpha_i$ and $y_i + x_i \cdot \tan\alpha_i$ are easy to calculate,
- $\alpha_0 + \alpha_1 + \alpha_2 + \cdots$ tends toward z.

As regards the first condition, α_i is chosen as follows: $\alpha_i = \tan^{-1}2^{-i}$ or $-\tan^{-1}2^{-i}$, so that $\tan\alpha_i = 2^{-i}$ or -2^{-i}. In order to satisfy the second condition, α_i is chosen in function of the difference $d_i = z - (\alpha_0 + \alpha_1 + \cdots + \alpha_{i-1})$: if $d_i < 0$, then $\alpha_i = \tan^{-1}2^{-i}$; else $\alpha_i = -\tan^{-1}2^{-i}$. The initial values are

$$x_0 = 1/k, \text{ where } k = (1+1)^{0.5}(1+2^{-2})^{0.5} \ldots (1+2^{-2(n-1)})^{0.5}, y_0 = 0,\ d_0 = z.$$

Thus, if $z \cong \alpha_0 + \alpha_1 + \ldots + \alpha_{n-1}$, then $x_n \cong \cos z$ and $y_n \cong \sin z$. It can be shown that the error is less than 2^{-n}. In the following algorithm, x_0 has been computed with $n = 16$ and 32 fractional bits.

Algorithm 10.13: CORDIC, $x = \cos z$, $y = \sin z$

```
x0 := 0.607252935; y0 := 0; d0 := z;
for i in 0 to n-1 loop
  if di < 0 then
    di+1 := di + tan-12-i; xi+1 := xi + yi·2-i; yi+1 := yi - xi·2-i;
  else
    di+1 := di - tan-12-i; xi+1 := xi - yi·2-i; yi+1 := yi + xi·2-i;
  end if;
end loop;
x := xn; y := yn;
```

A circuit for executing Algorithm 10.13 is shown in Fig. 10.12. It can be described by the following VDL model.

```
a <= angles(count)(31 DOWNTO 32 - m);
WITH d(m) SELECT next_d <=
  d + ('0'&a) WHEN '1', d - ('0'&a) WHEN OTHERS;
shifter_x: shifter GENERIC MAP(m => m)
PORT MAP(a => x, shift => count, b => shifted_x);
shifter_y: shifter GENERIC MAP(m => m)
PORT MAP(a => y, shift => count, b => shifted_y);
WITH d(m) SELECT next_x <=
  x + shifted_y WHEN '1', x - shifted_y WHEN OTHERS;
WITH d(m) SELECT next_y <=
  y - shifted_x WHEN '1', y + shifted_x WHEN OTHERS;
register_d: PROCESS(clk) ...
register_x: PROCESS(clk) ...
cos <= x(m DOWNTO m-p);
register_y: PROCESS(clk) ...
sin <= y(m DOWNTO m-p);
```

angles is a constant array defined within a user package; it stores $\tan^{-1}2^{-i}$, for i up to 15, with 32 bits:

```
CONSTANT angles: table := (
 x"c90fdaa2",
 x"76b19c15",
 x"3eb6ebf2",
 x"1fd5ba9a",
 x"0ffaaddb",
 x"07ff556e",
 x"03ffeaab",
 x"01fffd55",
 x"00ffffaa",
 x"007ffff5",
 x"003ffffe",
 x"001fffff",
 x"000fffff",
 x"0007ffff",
 x"0003ffff",
 x"0001ffff");
```

shifter is a previously defined component that computes $b = a \cdot 2^{-shift}$. A complete VHDL model *cordic2.vhd* is available at the Authors' web page. It includes an *n*-state counter, which generates the index *i* of Algorithm 10.13, and a control unit.

CORDIC can be used for computing other functions. In fact, Algorithm 10.13 is based on *circular* CORDIC *rotations*, defined in such a way that the difference $d_i = z - (\alpha_0 + \alpha_1 + \ldots + \alpha_{i-1})$ tends to 0. Another CORDIC mode, called *circular vectoring*, can be used. As an example, assume that at each step the value of α_i is chosen in such a way that y_i tends toward 0: if $sign(x_i) = sign(y_i)$, then $\boldsymbol{\alpha_i} = -\tan^{-1}2^{-i}$; else, $\boldsymbol{\alpha_i} = \tan^{-1}2^{-i}$. Thus, if $y_n \cong 0$, then x_n is the length of the initial vector multiplied by *k*. The following algorithm computes $(x^2 + y^2)^{0.5}$.

Algorithm 10.14: CORDIC, $z = (x^2 + y^2)^{0.5}$

```
x0 := x; y0 := y;
for i in 0 to n-1 loop
  if sign(xi) = sign(yi) then
    xi+1 := xi + yi·2^-i; yi+1 := yi - xi·2^-i;
  else
    xi+1 := xi - yi·2^-i; yi+1 := yi + xi·2^-i;
  end if;
end loop;
z := 0.607252935·xn;
```

Table 10.1 Binary to decimal converters

n	m	FFs	LUTs	Period	Total time
8	3	27	29	1.73	15.6
16	5	43	45	1.91	32.5
24	8	54	56	1.91	47.8
32	10	82	82	1.83	60.4
48	15	119	119	1.83	89.7
64	20	155	155	1.83	119.0

Table 10.2 Decimal-to-binary converters

n	m	FFs	LUTs	Period	Total time
8	3	26	22	1.80	16.2
16	5	43	30	1.84	31.3
24	8	65	43	1.87	46.8
32	10	81	51	1.87	61.7
48	15	118	72	1.87	91.6
64	20	154	92	1.87	121.6

A complete VHDL model *norm_cordic.vhd* corresponding to the previous aslgorithm is available at the Authors' web page.

10.7 FPGA Implementations

Several circuits have been implemented within a Virtex 5-2 device. The times are expressed in *ns* and the costs in numbers of Look Up Tables (LUTs) and flip-flops (FFs). All VHDL models are available at the Authors' web page.

10.7.1 Converters

Table 10.1 gives implementation results of several binary-to-decimal converters. They convert n-bit numbers to m-digit numbers.

In the case of decimal-to-binary converters, the implementation results are given in Table 10.2.

Table 10.3 Square rooters: restoring algorithm

n	FFs	LUTs	Period	Total time
8	38	45	2.57	20.6
16	71	79	2.79	44.6
24	104	113	3.00	72.0
32	136	144	3.18	101.8

Table 10.4 Square rooters: non-restoring algorithm

n	FFs	LUTs	Period	Total time
8	39	39	2.61	20.9
16	72	62	2.80	44.8
24	105	88	2.98	71.5
32	137	111	3.16	101.1

Table 10.5 Square rooter: Newton–Raphson method

n	p	FFs	LUTs	Period
8	0	42	67	2.94
8	4	51	78	3.50
8	8	59	90	3.57
16	8	92	135	3.78
16	16	108	160	3.92
32	16	173	249	4.35
32	32	205	301	4.67

Table 10.6 Base-2 logarithm

n	p	FFs	LUTs	DSPs	Period	Total time
8	10	16	20	1	4.59	45.9
16	18	25	29	1	4.59	82.6
24	27	59	109	2	7.80	210.5
32	36	44	46	4	9.60	345.6

10.7.2 Square Rooters

Three types of square rooters have been considered, based on the restoring algorithm (Fig. 10.3), the non-restoring algorithm (Fig. 10.4) and the Newton–Raphson method (Fig. 10.6). The implementation results are given in Tables 10.3, 10.4.

In the case of the Newton–Raphson method, the total time is data dependent. In fact, as was already indicated above, this type of circuit is generally not time effective (Table 10.5).

Table 10.7 Exponential 2^x

n	p	m	FFs	LUTs	DSPs	Period	Total time
8	8	13	27	29	1	4.79	38.3
16	16	23	46	48	2	6.42	102.7

Table 10.8 Exponential 2^x, version 2

n	p	m	k	FFs	LUTs	DSPs	Period	Total time
8	8	13	21	49	17	3	5.64	45.1
16	16	23	39	86	71	10	10.64	170.2

Table 10.9 CORDIC: sine and cosine

n	p	m	FFs	LUTs	Period	Total time
16	8	16	57	134	3.58	57,28
32	16	32	106	299	4.21	134.72
32	24	32	106	309	4.21	134.72

Table 10.10 CORDIC: $z = (x^2 + y^2)^{0.5}$

n	p	m	FFs	LUTs	DSPs	Period	Total time
8	8	16	43	136	1	3.39	27.12
16	16	32	76	297	2	4.44	71.04
48	24	48	210	664	5	4.68	224.64

10.7.3 Logarithm and Exponential

Table 10.6 gives implementation results of the circuit of Fig. 10.8. DSP slices have been used.

The circuit of Fig. 10.9 and the alternative circuit using a multiplier instead of a table (Fig. 10.10) have been implemented. In both cases DSP slices have been used (Tables 10.7, 10.8)

10.7.4 Trigonometric Functions

Circuits corresponding to algorithms 10.13 and 10.14 have been implemented. The results are summarized in Tables 10.9, 10.10.

10.8 Exercises

1. Generate VHDL models of combinational binary-to-decimal and decimal-to-binary converters.
2. Synthesize binary-to-radix-60 and radix-60-to-binary converters using LUT-6.

3. Implement Algorithm 10.6.
4. Implement the second square rooting convergence algorithm (based on Fig. 10.7).
5. Synthesize circuits for computing ln, log_{10}, e^x and10^x.
6. Generate a circuit which computes $e^{jx} = cos\,x + j \cdot sin\,x$.

References

1. Parhami B (2000) Computer arithmetic: algorithms and hardware design. Oxford University Press, New York
2. Volder JE (1959) The CORDIC trigonometric computing technique. IRE Trans Electron Comput EC8:330–334
3. Volder JE (2000) The birth of CORDIC. J VLSI Signal Process Sys 25:101–105