Appendix A

VHDL Module Description Code

This appendix contains all the VHDL entities and architectures required to get the fully functional versions of the designs presented in Chapter 3. Both versions require these libraries, modules and functions:

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.math_real.log2;
use ieee.math_real.ceil;
```

There are many common modules between both versions. These modules are only included in the SPA vulnerable version.

A.1 SPA Vulnerable Version

A.1.1 Modular Exponentiation

This entity is the top one of the hierarchy.

```vhdl
entity modexp is
    generic(n:integer);
    port(
        rst, CLK0, CLK1, CLK2 :in std_logic;
        alfa, beta, u, modu2n, modu22n :in std_logic_vector(n-1 downto 0);
        e :out std_logic_vector(n-1 downto 0)
    );
end;
```

architecture arch of modexp is
signal S0 : std_logic_vector(1 downto 0);
signal S1, C12, alfa_i, EN0 : std_logic;
signal x, y, r, r_e, beta_t, e_t, one : std_logic_vector(n-1 downto 0);
signal aux_one : std_logic_vector(n-2 downto 0);

begin

aux_one <= (others => '0');
one <= aux_one & "1";

ins_c12main: entity work.c12main
  generic map(log2n => integer(ceil(log2(real(n))))))
  port map(rst, CLK0, CLK2, C12);
ins_s0: entity work.s0
  generic map(n=>n)
  port map(rst, CLK0, CLK2, C12, alfa, S0, alfa_i);
ins_s1: entity work.s1
  port map(rst, CLK0, CLK2, S1);
ins_mp: entity work.mp
  generic map(n=>n)
  port map(CLK0, CLK1, CLK2, EN0, x, y, u, r);

for gen_0: for i in 0 to n/4-1 generate
  x(i) <= (beta(i) and (not(S0(1)) and not(S0(0)))) or (e_t(i) and (S 0(1) or S0(0)))
  y(i) <= (modu22n(i) and (not(S0(1)) and not(S0(0)))) or (e_t(i) and (not(S0(1)) and S0(0))) or (beta_t(i) and (not(S0(0)) and S0(1))) o
  r_e(i) <= (r(i) and S1) or (modu2n(i) and not(S1));
end generate;
for gen_1: for i in n/4 to n/2-1 generate ... 
for gen_2: for i in n/2 to 3*n/4-1 generate ...
for gen_3: for i in 3*n/4 to n-1 generate ... 

process (CLK2) begin
  if rising_edge(CLK2) then
    if CLK0 = '1' then
      ENO <= rst or not(ENO) or not(ENO and not(alfa_i) and S0(0) and
A.1. SPA Vulnerable Version

not(S0(1)))
if EN0 = '1' then e_t <= r_e; end if;
if S1 = '0' then beta_t <= r; end if;
if S0 = "11" then e <= r; end if;
end if;
end if;
end process;
end arch;

A.1.2 Montgomery Product

entity mp is
    generic(n:integer);
    port(
        CLK0, CLK1, CLK2, EN0 :in std_logic;
        x, y, u :in std_logic_vector(n-1 downto 0);
        r_n :out std_logic_vector(n-1 downto 0)
    );
end;

architecture arch of mp is

signal S2, S3, x_i, c_out_0, c_out_1, sign, C12 : std_logic;
signal b, aux_b : std_logic_vector(n-1 downto 0);
signal a, f, s, r : std_logic_vector(n+1 downto 0);
signal x_sr : std_logic_vector(n-2 downto 0);

begin

ins_c12: entity work.c12
    generic map(log2n => integer(ceil(log2(real(n)))))
    port map(CLK0, CLK1, CLK2, C12);
ins_s2: entity work.s2
    port map(CLK0, CLK2, S2);
ins_s3: entity work.s3
    port map(CLK0, CLK1, CLK2, S3);
ins_ppa_0: entity work.ppa
    generic map(n=>2)
port map(a(1 downto 0), b(1 downto 0), C12, C12, s(1 downto 0), c_o ut_0);

ins_ppa_1: entity work.ppa
generic map(n=>n)
port map(a(n+1 downto 2), aux_b, c_out_0, C12, s(n+1 downto 2), c_o ut_1);

for_b_0: for i in 0 to n/4-1 generate
b(i) <= (not(S2 or C12) and x_i and y(i)) or ((S2 or C12) and (C12 or f(0)) and u(i));
end generate;
for_b_1: for i in n/4 to n/2-1 ...
for_b_2: for i in n/2 to 3*n/4-1 ...
for_b_3: for i in 3*n/4 to n-1 ...

for_a_0: for i in 0 to n/4-1 generate
a(i) <= ((not(S2) or C12) and f (i+1)) or (S2 and not(C12) and f(i));
end generate;
for_a_1: for i in n/4 to n/2-1 ...
for_a_2: for i in n/2 to 3*n/4-1 ...
for_a_3: for i in 3*n/4 to n-1 ...
for_a_4: for i in n to n ...

for_r_0: for i in 0 to n/4-1 generate
r(i) <= (sign and C12) and a(i)) or (not(sign and C12) and s(i));
end generate;
for_r_1: for i in n/4 to n/2-1 ...
for_r_2: for i in n/2 to 3*n/4-1 ...
for_r_3: for i in 3*n/4 to n-1 ...
for_r_4: for i in n to n+1 ...

a(n+1) <= f(n+1) and S2 and not(C12);
aux_b <= "00" & b(n-1 downto 2);
sign <= s(n+1);

process (CLK2) begin
if rising_edge(CLK2) then
if CLK0 = '1' then
x_sr <= (others => '0');
elsif CLK0 = '0' then
if CLK1 = '1' and EN0 = '1' then
if S3 = '0' then
x_i <= x(0);
A.1. SPA Vulnerable Version

\[
x_{sr} \leq x(n-1 \text{ downto } 1);
f \leq \text{(others } \Rightarrow '0') \text{;}
\]
else if S3 = '1' then
\[
x_i \leq x_{sr}(0);
x_{sr} \leq '0' \& x_{sr}(n-2 \text{ downto } 1);
f \leq r;
\]
end if;
else
\[
\text{if } C12 = '0' \text{ then if S3 = '1' then } f \leq r; \text{ end if;}
\text{else } r_n \leq r(n-1 \text{ downto } 0); \text{ end if;}
\text{end if;}
\text{end if;}
\text{end if;}
\text{end process;}
\]
end if;

end arch;

A.1.3 Adder/Subtractor

entity ppa is
generic(n:integer);
port(
a, aux_b : in std_logic_vector(n-1 downto 0);
c_in, sign : in std_logic;
s : out std_logic_vector(n-1 downto 0);
c_out : out std_logic
);
end;
architecture arch of ppa is
signal b, g, p, gen_g, gen_p : std_logic_vector(n-1 downto 0);
signal c : std_logic_vector(n downto 0);
begin

ins_dp: entity work.dp
generic map(n=>n)
port map(g, p, gen_g, gen_p);
for_all_0: for i in 0 to n/4-1 generate
  b(i) <= (not(aux_b(i)) and sign) or (aux_b(i) and not(sign));
  g(i) <= a(i) and b(i);
  p(i) <= a(i) or b(i);
  c(i+1) <= gen_g(i) or (gen_p(i) and c_in);
  s(i) <= (a(i) xor b(i)) xor c(i);
end generate;
for_all_1: for i in n/4 to n/2-1 ...
for_all_2: for i in n/2 to 3*n/4-1 ...
for_all_3: for i in 3*n/4 to n-1 ...
  c(0) <= c_in;
  c_out <= c(n);
end arch;

A.1.4 Dot Procedure

entity dp is
  generic(n:integer);
  port(
    g, p :in std_logic_vector(n-1 downto 0);
    gen_g, gen_p :out std_logic_vector(n-1 downto 0)
  );
end;

architecture arch of dp is

signal aux_g_0, aux_g_1, aux_p_0, aux_p_1 : std_logic_vector((n/2-1) downto 0);

begin

  if_e2: if n = 2 generate
    gen_g(0) <= g(0);
    gen_p(0) <= p(0);
    gen_g(1) <= g(1) or (p(1) and g(0));
    gen_p(1) <= p(1) and p(0);
  end generate;
A.1. SPA Vulnerable Version

end generate;

if_g2: if n > 3 generate
ins_dp_low: entity work.dp
generic map(n=>n/2)
port map(g((n/2-1) downto 0), p((n/2-1) downto 0), aux_g_0, aux_p_0);

ins_dp_high: entity work.dp
generic map(n=>n/2)
port map(g((n-1) downto (n/2)), p((n-1) downto (n/2)), aux_g_1, aux_p_1);

for_high_0: for i in 0 to (n/4-1) generate
    gen_g(i+n/2) <= (aux_p_1(i) and aux_g_0(n/2-1)) or aux_g_1(i);
    gen_p(i+n/2) <= aux_p_1(i) and aux_p_0(n/2-1);
end generate;

for_high_1: for i in n/4 to (n/2-1) ...

    gen_g((n/2-1) downto 0) <= aux_g_0;
    gen_p((n/2-1) downto 0) <= aux_p_0;
end generate;

end arch;

A.1.5 Main Counter $C_{12}$

entity c12main is
generic(log2n:integer);
port(
    rst, CLK0, CLK2 :in std_logic;
    C12 :out std_logic
);
end;

architecture arch of c12main is

signal c : std_logic_vector(log2n downto 0);
signal CLK00 : std_logic;
Appendix A. VHDL Module Description Code

begin

C12 <= c(log2n) and c(0);

process (CLK2) begin
    if rising_edge(CLK2) then
        if CLK0 = '1' then
            if rst = '1' then
                c <= (others => '0');
                CLK00 <= '0';
            else
                if CLK00 = '1' then CLK00 <= '0';
                else c <= c + '1'; CLK00 <= '1'; end if;
            end if;
        end if;
    end if;
end process;
end arch;

A.1.6 MP’s Counter $C_{12}'$

entity c12 is
    generic (log2n:integer);
    port(
        CLK0, CLK1, CLK2 :in std_logic;
        C12 :out std_logic
    );
end;

architecture arch of c12 is

signal c : std_logic_vector(log2n downto 0);

begin

C12 <= c(log2n) and c(0);

process (CLK2) begin

if rising_edge(CLK2) then
    if CLK0 = '1' then
        c <= (others => '0');
    else
        if CLK1 = '1' then
            c <= c + '1';
        end if;
    end if;
end if;
end process;

end arch;

A.1.7 S0 Generator

entity s0 is
    generic(n:integer);
    port(
        rst, CLK0, CLK2, C12 :in std_logic;
        alfa :in std_logic_vector(n-1 downto 0);
        S0 :out std_logic_vector(1 downto 0);
        alfa_i_out :out std_logic
    );
end;

architecture arch of s0 is

    type state_type is (e0, e1, e2, e3);
    signal state : state_type;
    signal alfa_i, aux : std_logic;

    begin

        alfa_i_out <= alfa_i;

        ins_alfa: entity work.alfa
            generic map(n=>n)
            port map(rst, CLK0, CLK2, alfa, alfa_i);
process (CLK2) begin
  if rising_edge(CLK2) then
    if CLK0 = '1' then
      if rst = '1' then
        state <= e0;
        aux <= '0';
      else
        case state is
          when e0 =>
            if aux = '0' then
              state <= e0;
              aux <= '1';
            elsif aux = '1' then
              state <= e1;
            end if;
          when e1 => state <= e2;
          when e2 =>
            if C12 = '0' then state <= e1;
            else state <= e3; end if;
          when e3 => state <= e3;
        end case;
      end if;
    else
      case state is
        when e0 => S0 <= "00";
        when e1 => S0 <= "01";
        when e2 => S0 <= "10";
        when e3 => S0 <= "11";
      end case;
    end if;
  end if;
end process;

process (state) begin
  case state is
    when e0 => S0 <= "00";
    when e1 => S0 <= "01";
    when e2 => S0 <= "10";
    when e3 => S0 <= "11";
  end case;
end process;

end arch;
A.1.8 S1 Generator

entity s1 is
port(
    rst, CLK0, CLK2 :in std_logic;
    S1 :out std_logic
);
end;

architecture arch of s1 is

type state_type is (e0, e1);
signal state : state_type;
signal aux : std_logic;

begin

process (CLK2) begin
    if rising_edge(CLK2) then
        if CLK0 = '1' then
            if rst = '1' then
                state <= e0;
                aux <= '0';
            else
                case state is
                    when e0 =>
                        if aux = '0' then
                            state <= e0;
                            aux <= '1';
                        elsif aux = '1' then
                            state <= e1;
                        end if;
                    when e1 => state <= e1;
                    end case;
            end if;
        end if;
    end if;
end process;
process (state) begin
  case state is
    when e0 => S1 <= '0';
    when e1 => S1 <= '1';
  end case;
end process;
end arch;

A.1.9 S2 Generator

entity s2 is
  port(
    CLK0, CLK2 :in std_logic;
    S2 :out std_logic
  );
end;

architecture arch of s2 is

  type state_type is (e0, e1);
  signal state : state_type;

begin

  process (CLK2) begin
    if rising_edge(CLK2) then
      if CLK0 = '1' then
        state <= e0;
      else
        case state is
          when e0 => state <= e1;
          when e1 => state <= e0;
        end case;
      end if;
    end if;
    end if;
  end process;

  process (state) begin
    case state is
    when e0 => S1 <= '0';
    when e1 => S1 <= '1';
  end case;
end process;
end arch;
case state is
    when e0 => S2 <= '0';
    when e1 => S2 <= '1';
end case;
end process;

end arch;

A.1.10 S3 Generator

entity s3 is
    port(
        CLK0, CLK1, CLK2 :in std_logic;
        S3 :out std_logic
    );
end;

architecture arch of s3 is

type state_type is (e0, e1);
signal state : state_type;

begin

    process (CLK2) begin
        if rising_edge(CLK2) then
            if CLK0 = '1' then
                state <= e0;
            else
                if CLK1 = '1' then
                    case state is
                        when e0 => state <= e1;
                        when e1 => state <= e1;
                    end case;
                end if;
            end if;
        end if;
    end process;

end arch;
process (state) begin
  case state is
    when e0 => S3 <= '0';
    when e1 => S3 <= '1';
  end case;
end process;

end arch;

A.1.11 \( \alpha_i \) Generator

entity alfa is
generic(n:integer);
port(
  rst, CLK0, CLK2 :in std_logic;
  alfa :in std_logic_vector(n-1 downto 0);
  alfa_i :out std_logic
);
end;

architecture arch of alfa is

signal aux_alfa_i, CLK00, aux : std_logic;
signal alfa_sr : std_logic_vector(n-1 downto 0);

begin
  alfa_i <= aux_alfa_i;

  process (CLK2) begin
    if rising_edge(CLK2) then
      if CLK0 = '1' then
        if rst = '1' then
          alfa_sr <= alfa;
          CLK00 <= '0';
          aux <= '0';
        else
          if aux = '0' then aux <= '1';
        end if;
      end if;
    end if;
  end process;

  alfa_sr <= alfa;
end arch;
A.2. SPA Resistant Version

A.2.1 Modular Exponentiation

This entity is the top one of the heirachy.

declaration modexp is
    generic(n:integer);
    port(  
        rst, CLK0, CLK1, CLK2 :in std_logic;
        alfa, beta, u, modu2n, modu22n :in std_logic_vector(n-1 downto 0);
        e :out std_logic_vector(n-1 downto 0)
    );
end;

architecture arch of modexp is

signal S0 : std_logic_vector(1 downto 0);
signal S1, C12, EN1 : std_logic;
signal x, y, r, r_e, beta_t, e_t, one : std_logic_vector(n-1 downto 0);
signal aux_one : std_logic_vector(n-2 downto 0);

begin

    aux_one <= (others => '0');
Appendix A. VHDL Module Description Code

one <= aux_one & "1";

ins_c12main: entity work.c12main
    generic map(log2n => integer(ceil(log2(real(n)))))
    port map(rst, CLK0, CLK2, EN1, C12);
ins_s0: entity work.s0
    generic map(n=>n)
    port map(rst, CLK0, CLK2, C12, alfa, S0, EN1);
ins_s1: entity work.s1
    port map(rst, CLK0, CLK2, S1);
ins_mp: entity work.mp
    generic map(n=>n)
    port map(CLK0, CLK1, CLK2, x, y, u, r);

for_gen_0: for i in 0 to n/4-1 generate
    x(i) <= (beta(i) and (not(S0(1)) and not(S0(0)))) or (e_t(i) and (S0(1) or S0(0)));
    y(i) <= (modu22n(i) and (not(S0(1)) and not(S0(0)))) or (e_t(i) and (not(S0(1)) and S0(0))) or (beta_t(i) and (not(S0(0)) and S0(1))) or (one(i) and (S0(1) and S0(0)));
    r_e(i) <= (r(i) and S1) or (modu2n(i) and not(S1));
end generate;
for_gen_1: for i in n/4 to n/2-1 ...
for_gen_2: for i in n/2 to 3*n/4-1 ...
for_gen_3: for i in 3*n/4 to n-1 ...

process (CLK2) begin
    if rising_edge(CLK2) then
        if CLK0 = '1' then
            e_t <= r_e;
            if S1 = '0' then beta_t <= r; end if;
            if S0 = "11" then e <= r; end if;
        end if;
    end if;
end process;
end arch;
A.2.2 Montgomery Product

text

entity mp is
  generic(n:integer);
  port(
    CLK0, CLK1, CLK2 : in std_logic;
    x, y, u : in std_logic_vector(n-1 downto 0);
    r_n : out std_logic_vector(n-1 downto 0)
  );
end;

architecture arch of mp is

signal S2, S3, x_i, c_out_0, c_out_1, sign, C12 : std_logic;
signal b, aux_b : std_logic_vector(n-1 downto 0);
signal a, f, s, r : std_logic_vector(n+1 downto 0);
signal x_sr : std_logic_vector(n-2 downto 0);

begin

  ins_c12: entity work.c12
    generic map(log2n => integer(ceil(log2(real(n)))))
    port map(CLK0, CLK1, CLK2, C12);
  ins_s2: entity work.s2
    port map(CLK0, CLK2, S2);
  ins_s3: entity work.s3
    port map(CLK0, CLK1, CLK2, S3);
  ins_ppa_0: entity work.ppa
    generic map(n=>2)
    port map(a(1 downto 0), b(1 downto 0), C12, C12, s(1 downto 0), c_out_0);
  ins_ppa_1: entity work.ppa
    generic map(n=>n)
    port map(a(n+1 downto 2), aux_b, c_out_0, C12, s(n+1 downto 2), c_out_1);

  for_b_0: for i in 0 to n/4-1 generate b(i) <= (not(S2 or C12) and x_i
    and y(i)) or ((S2 or C12) and (C12 or f(0)) and u(i)); end generate;
  for_b_1: for i in n/4 to n/2-1 ...


for_b_2: for i in n/2 to 3*n/4-1 ...
for_b_3: for i in 3*n/4 to n-1 ...
for_a_0: for i in 0 to n/4-1 generate a(i) <= ((not(S2) or C12) and f(i+1)) or (S2 and not(C12) and f(i)); end generate;
for_a_1: for i in n/4 to n/2-1 ...
for_a_2: for i in n/2 to 3*n/4-1 ...
for_a_3: for i in 3*n/4 to n-1 ...
for_a_4: for i in n to n ...
for_r_0: for i in 0 to n/4-1 generate r(i) <= ((sign and C12) and a(i)) or (not(sign and C12) and s(i)); end generate;
for_r_1: for i in n/4 to n/2-1 ...
for_r_2: for i in n/2 to 3*n/4-1 ...
for_r_3: for i in 3*n/4 to n-1 ...
for_r_4: for i in n to n+1 ...
a(n+1) <= f(n+1) and S2 and not(C12);
aux_b <= "00" & b(n-1 downto 2);
sign <= s(n+1);

process (CLK2) begin
if rising_edge(CLK2) then
if CLK0 = '1' then
x_sr <= (others => '0');
elsif CLK0 = '0' then
if CLK1 = '1' then
if S3 = '0' then
x_i <= x(0);
x_sr <= x(n-1 downto 1);
f <= (others => '0');
elif S3 = '1' then
x_i <= x_sr(0);
x_sr <= '0' & x_sr(n-2 downto 1);
f <= r;
end if;
else
if C12 = '0' then if S3 = '1' then f <= r; end if;
else r_n <= r(n-1 downto 0); end if;

A.2. SPA Resistant Version

A.2.3 Main Counter $C_{12}$

entity c12main is
generic(log2n:integer);
port(
  rst, CLK0, CLK2, EN1 :in std_logic;
  C12 :out std_logic
);
end;

architecture arch of c12main is

signal c : std_logic_vector(log2n downto 0);

begin

  C12 <= c(log2n) and c(0);

  process (CLK2) begin
    if rising_edge(CLK2) then
      if CLK0 = '1' then
        if rst = '1' then
          c <= (others => '0');
        elsif EN1 = '1' then
          c <= c + '1';
        end if;
      end if;
    end if;
  end process;

end arch;
### A.2.4 S0 Generator

```vhdl
entity s0 is
    generic (n: integer);
    port(
        rst, CLK0, CLK2, C12 : in std_logic;
        alfa : in std_logic_vector (n-1 downto 0);
        S0 : out std_logic_vector (1 downto 0);
        EN1 : out std_logic
    );
end;
architecture arch of s0 is

type state_type is (e0, e1, e2, e3);
signal state : state_type;
signal alfa_i, aux : std_logic;

begin

    ins_alfa: entity work.alfa
        generic map(n=>n)
        port map(rst, CLK0, CLK2, alfa, alfa_i, EN1);

    process (CLK2) begin
        if rising_edge(CLK2) then
            if CLK0 = '1' then
                if rst = '1' then
                    state <= e0;
                    aux <= '0';
                else
                    case state is
                        when e0 =>
                            if aux = '0' then
                                state <= e0;
                                aux <= '1';
                            elsif aux = '1' then
                                state <= e1;
                            end if;
            end if;
        end if;
    end process;
end;
```

Appendix A. VHDL Module Description Code
A.2. SPA Resistant Version

when e1 =>
    if alfa_i = '1' then state <= e2;
    else
        if C12 = '0' then state <= e1;
        else state <= e3; end if;
    end if;
when e2 =>
    if C12 = '0' then state <= e1;
    else state <= e3; end if;
when e3 => state <= e3;
end case;
end if;
end if;
end if;
end process;

process (state) begin
    case state is
        when e0 => S0 <= "00";
        when e1 => S0 <= "01";
        when e2 => S0 <= "10";
        when e3 => S0 <= "11";
    end case;
end process;
end arch;

A.2.5 $\alpha_i$ Generator

entity alfa is
    generic(n:integer);
    port(  
        rst, CLK0, CLK2 :in std_logic;
        alfa :in std_logic_vector(n-1 downto 0);
        alfa_i, EN1 :out std_logic  
    );
end;

architecture arch of alfa is
Appendix A. VHDL Module Description Code

```vhdl
signal aux_alfa_i, z, aux : std_logic;
signal alfa_sr : std_logic_vector(n-1 downto 0);

begin

  EN1 <= not(not(z) and aux_alfa_i);
  alfa_i <= aux_alfa_i;

  process (CLK2) begin
    if rising_edge(CLK2) then
      if CLK0 = '1' then
        if rst = '1' then
          alfa_sr <= alfa;
          z <= '1';
          aux <= '0';
        else
          if aux = '0' then aux <= '1';
          else
            if not(not(z) and aux_alfa_i) = '1' then
              aux_alfa_i <= alfa_sr(n-1);
              alfa_sr <= alfa_sr(n-2 downto 0) & "0";
            end if;
            z <= aux_alfa_i and not(z);
          end if;
        end if;
      end if;
    end if;
  end process;
end arch;
```

A.3 Test Bench

The script below corresponds to the test bench module for the simulation with gate delays for both 16-bit versions.

```vhdl
entity tb_modexp is end;
```
architecture arch of tb_modexp is

signal rst, CLK0, CLK1, CLK2 : std_logic;
signal alfa, beta, u, e, modu2n, modu22n : std_logic_vector(15 downto 0);

begin

ins_modexp: entity work.modexp_n16
  port map(rst, CLK0, CLK1, CLK2, alfa, beta, u, modu2n, modu22n, e);

beta <= "1010001000101010";
alfa <= "0101110111110011";
u <= "1010111110000001";
modu2n <= "0101000001111111";
modu22n <= "0101111110010110";

process begin
  rst <= '0'; wait for 7100 ns;
  rst <= '1'; wait for 200 ns;
  rst <= '0'; wait;
end process;

process begin
  CLK0 <= '0'; wait for 7150 ns;
  for i in 0 to 5000 loop
    CLK0 <= '1'; wait for 100 ns;
    CLK0 <= '0'; wait for 7100 ns;
  end loop;
end process;

process begin
  CLK1 <= '0'; wait for 7150 ns;
  for i in 0 to 90000 loop
    CLK1 <= '1'; wait for 100 ns;
    CLK1 <= '0'; wait for 300 ns;
  end loop;
end process;

process begin
CLK2 <= '1'; wait for 100 ns;
CLK2 <= '0'; wait for 100 ns;
end process;

end arch;
Appendix B

Scripts and Commands for Cadence® Software

This appendix contains the structures of the scripts and the commands needed to retrieve the results presented in Chapter 4.

B.1 Simulation

This section includes scripts for simulating with and without gate delays and commands for obtaining the power traces.

B.1.1 Simulation without Gate Delays

The first commands are shell commands to create a working directory where libraries are placed. Later, the VHDL description is read, elaborated and simulated.

```
$ if [ ! -d work ]; then mkdir work fi
$ if [ ! -d hdl.var ]; then touch hdl.var fi
$ if [ ! -f cds.lib ]; then cat << 'EOF' > cds.lib
include $CDS_INST_DIR/tools/inca/files/cds.lib
define work work
EOF fi

$ ncvhdl -work work -v93 \ testbench.vhd \ mp.vhd \ ...
$ ncelab -access +rwc -work work testbench
$ ncsim -gui testbench
```
B.1.2 Simulation with Gate Delays

The first commands are shell commands to create a working directory where libraries are placed. Later, the delays file .sdf is read and compiled, the Verilog synthesized description is read, as well as the CMOS library ($CORE65). Finally the top entity is elaborated and simulated.

```bash
if [ ! -d work ]; then mkdir work fi
if [ ! -d hdl.var ]; then touch hdl.var fi
if [ ! -f cds.lib ]; then cat << 'EOF' > cds.lib
include $CDS_INST_DIR/tools/inca/files/cds.lib
define work work EOF fi

ncsdfc -compile -cdslib cds.lib modexp.sdf

rm -f sdf.cmd
cat << 'EOF' >> sdf.cmd
COMPILED_SDF_FILE=modexp.sdf.X, SCOPE=ins_modexp, MTM_CONTROL=M AXIMUM'; EOF

ncvlog -work work $CORE65 modexp.v
ncvhdl -work work -v93 testbench.vhd
ncelab -sdf_cmd_file sdf.cmd -access +rwc -work work testbench
ncsim -gui testbench
```

B.1.3 Power Consumption Analysis

First of all, in the ncsim console, the next commands have to be run to get the nodal activity in each period:

run $TIME_UNTIL_RESET

dumptcf -scope ins_modexp -output period_0.tcf -overwrite
run $PERIOD
dumptcf -end

dumptcf -scope ins_modexp -output period_1.tcf -overwrite
run $PERIOD
dumptcf -end

...
After getting all the .tcf files, *Encounter RTL Compiler* computes the average power consumption for each period:

```tcl
set_attribute lib_search_path $LIBPATH
set_attribute library $LIB

read_hdl modexp.v
elaborate

read_tcf period_0.tcf
report power > period_0.rpt

read_tcf period_1.tcf
report power > period_1.rpt

...  
```

Finally, using a generic programming language, all the reports have to be summarized to get all the average power consumption values. These values are on line 14 between columns 38 and 46 in the reports.

**B.2 Synthesis**

This section includes scripts for the synthesis using *Encounter RTL Compiler*.

**B.2.1 Flatten NOR2X2/IV/DFPQ Synthesis**

In order to simulate a design without preserving hierarchy, and only with NOR2X2 gates, inverters and D-type flip-flops, this script is needed:

```tcl
set_attribute lib_search_path $LIBPATH
set_attribute library $LIB

set_attribute avoid 1 [find / -libcell *]
set_attribute avoid 0 [find / -libcell *_NOR2X2*]
set_attribute avoid 1 [find / -libcell *_NOR2X25*]
set_attribute avoid 0 [find / -libcell *IVX*]
set_attribute avoid 0 [find / -libcell *DFPQX*]
set_attribute avoid 1 [find / -libcell *SDFPQX*]
```
read_hdl -vhdl $FILES
elaborate -parameters {$N} $TOP

define_clock -name $CLK -period $PERIOD [find / -port $CLK]
synthesize -to_mapped

ungroup -flatten -all

write_hdl -mapped > $TOP.v

In the script above, each parameter preceded by a $ symbol is a reference of the value, that must be set at the beginning of the script. In order to retrieve the delays file:

write_sdf -design modexp_n16 -precision 3 -edges check_edge -version 3.0 > $TOP.sdf