VHDL redactable scheme implementation

Romain Michard

November 9, 2017

This document aims at explaining the architecture and utilization of the VHDL component implementing a Merkle tree with the KECCAK hash function as it seems to be a good way to have a redactable scheme. KECCAK has been chosen because the NIST made it the SHA-3 standard so a good post-quantum hash function. Its hardware VHDL description code is given by the creators website[BDPA].

1 Node

The main component in a tree is the node so we start by explaining how it works.

This entity has to get two 256-bit hash words (the results of two other nodes), concatenate them as a 512-bit input message, hash this message and obtain a new 256-bit word that is output.

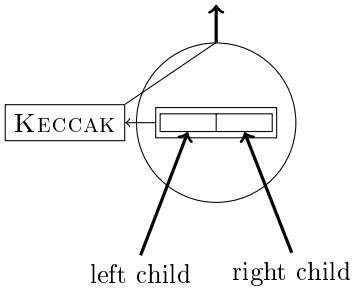

Figure 1: A node

The VHDL code is given in appendix A.

- Line 6: clock;
- Line 7: an asynchronous reset (active low);
- Line 8: when left child is done;
- Line 9: when right child is done;
- Line 10: when present node has finished;
- Line 11: left child hash value;
- Line 12: right child hash value;
- Line 13: present node hash result;
- Lines 19–25: ports communicating with the KECCAK module;
- Line 27: a signal to know if the KECCAK module is attributed to the present node;

- Lines 33–36: signals to control KECCAK;
- Lines 38–40: the state of the fsm (copied from the original KECCAK test bench);
- Line 42: a counter used in the fsm;
- Line 44: registers to memorize the state of the children;
- Lines 47–60: process to control these registers;
- Line 62: start only if both children are OK and the KECCAK module is attributed to this node;
- Lines 73–177: the process to control KECCAK adapted from the original test bench to conform with the inputs and the output.

2 Tree

From this node a tree can be built. A binary tree is represented in Figure 2. It's a logical representation which is tricky to make in VHDL. For simplification this tree is constructed as a two-dimensional matrix (see Figure 3). Say l is the number of leaves for the tree. So the matrix has $\log_2(l)$ rows and $\frac{l}{2}$ columns. Each node is represented by his coordinates x and y. For each row y ($1 \leq y \leq \log_2(l)$) we have $0 \leq x < 2^{y-1}$. A node can also be represented by an index $n = 2^{y-1} + x$ and its children are $2n$ and $2n + 1$. One can verify than $2^{y-1} \leq n \leq 2^y - 1$.

A simple package (Appendix B) has been required to have a type for the leaves and another one for the witness. The explanation of the tree code (Appendix C) is:

- Line 10: a generic integer to specify the leaves number (must be a power of 2);
- Line 11: log_2 of this number;
- Lines 14–15: classical control inputs;
- Line 16: a signal to tell the leaves are OK and the Merkle algorithm can start;
- Lines 17–18: the hash result;
- Line 19: the leaves as a predefined type;
- Line 20: the signal to tell the result is valid;
- Line 21: the witness defined as a `wit_type` signal (see the package in Appendix B where `256*3-1` at line 8 has to be replaced by `256*logl-1` when changing the number of leaves;

- Lines 65–67: hash results of the nodes in the tree;
- Lines 69–75: signals from each node to handle the KECCAK module when possible;
- Lines 77–80: signals to connect the KECCAK module;
- Lines 82–83: a counter to know which node can use the KECCAK module;
- Lines 85–86: register indicating the completion of the result parts;
- Lines 89–92: making the leaves appear as nodes;
- Lines 162–168: connecting the KECCAK module depending on the actual considered node;
- Lines 170–183: computing the tree outputs;
- Lines 185–199: decrementing the current node counter when the actual step is finished;
- Lines 201–208: allocating the KECCAK module to a node depending on `n_cntr`;
- Lines 210–238: computing the `witness` output from `s_hash` values.

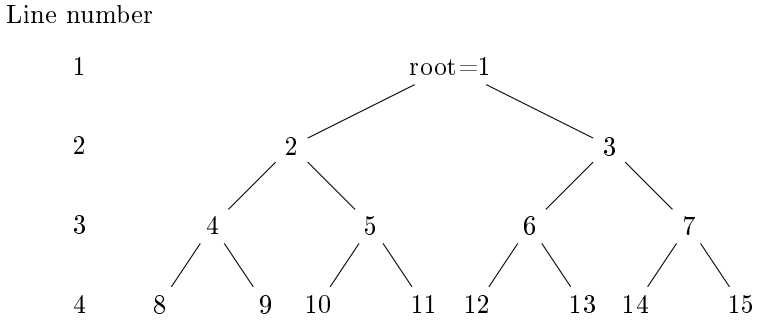

Figure 2: Tree representation

3 Witness

The *Witness* has to be output for any leaf to be able to compute a root then compare to the one that is signed.

For an example we can look at Figure 2. Say there are 8 leaves (from 0 to 7) and one want to check whether the considered file is indeed the leaf number 4 or not. The user has to hash this file then concatenate it with the number 5 hash result (part of the witness given to the user), hash that then

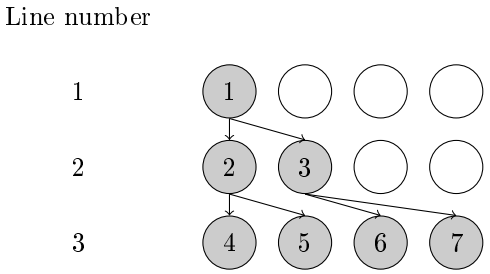

Figure 3: Tree implementation as a matrix

concatenate to the node 7 hash result (so another part of the witness), hash that, concatenate with the node 2 hash result (the last part of the witness), concatenate then finally compare to the signed root. In the component all the hash results are computed and stored in `s_hash(idx)` where `idx` is the node number (from 1 to `leaves_number-1`). So the witness of number 4 is `witness(5)=(s_hash(12)|s_hash(7)|s_hash(2))`.

References

[BDPA] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. The KECCAK sponge function family. http://keccak.noekeon.org/.

A merkle_node.vhd

```
library ieee;
use ieee.std_logic_1164.all;

entity merkle_node is
  port(
    clk          : in  std_logic;
    rst_n        : in  std_logic;
    start_l      : in  std_logic;
    start_r      : in  std_logic;
    node_done    : out std_logic := '0';
    n_l          : in  std_logic_vector(255 downto 0);
    n_r          : in  std_logic_vector(255 downto 0);
    node_out     : out std_logic_vector(255 downto 0)
                       := (others => '0' );

    ------------------------------
    --  keccak interface   --
    ------------------------------
    k_init       : out std_logic;
    k_absorb     : out std_logic;
    k_go         : out std_logic;
    k_squeeze    : out std_logic;
    k_ready      : in  std_logic;
    k_din        : out std_logic_vector(63 downto 0);
    k_dout       : in  std_logic_vector(63 downto 0);

    k_att        : in  std_logic
  );
end merkle_node;

architecture rtl of merkle_node is

  signal node_init,node_go,node_absorb,node_ready,
         node_squeeze,node_start,s_node_done : std_logic;
  signal node_din,node_dout :
         std_logic_vector(63 downto 0);

  type st_type is (initial,read_first_input,st0,st1,
                   st1a,END_HASH1,END_HASH2,stop_k);
  signal st : st_type;

  signal counter : integer range 0 to 15;
```

```
    signal r_start_l, r_start_r : std_logic;

begin
    start_registers : process(clk,rst_n)
    begin
      if rst_n = '0' then
        r_start_l <= '0';
        r_start_r <= '0';
      elsif clk'event and clk = '1' then
        if start_l = '1' then
          r_start_l <= '1';
        end if;
        if start_r = '1' then
          r_start_r <= '1';
        end if;
      end if;
    end process;

    node_start <= r_start_l and r_start_r and k_att;
    node_done <= s_node_done;

    k_init <= node_init;
    k_go <= node_go;
    k_absorb <= node_absorb;
    k_squeeze <= node_squeeze;
    k_din <= node_din;
    node_ready <= k_ready;
    node_dout <= k_dout;

    p_main : process(clk,rst_n)
    begin
      if rst_n = '0' then
        node_din <= (others=>'0');
        node_init <= '0';
        node_absorb <= '0';
        node_squeeze <= '0';
        node_go <= '0';
        counter <= 0;
        st <= initial;
        s_node_done <= '0';
        node_out <= (others => '0');

      elsif clk'event and clk = '1' then
        case st is
          when initial =>
            if(node_start='1') then
```

```
90              st <= read_first_input;
                node_init <= '1';
                counter <= 0;
              end if;

95          when read_first_input =>
              node_init<='0';
              if(counter=0) then
                node_din<=n_l(255 downto 192);
                node_absorb<='1';
100             st<=st0;
                counter <= 1;
              end if;

            when st0 =>
105           if(counter<9) then
                case counter is
                  when 1 =>
                    node_din <= n_l(191 downto 128);
                  when 2 =>
110                 node_din <= n_l(127 downto 64);
                  when 3 =>
                    node_din <= n_l(63 downto 0);
                  when 4 =>
                    node_din <= n_r(255 downto 192);
115               when 5 =>
                    node_din <= n_r(191 downto 128);
                  when 6 =>
                    node_din <= n_r(127 downto 64);
                  when others =>
120                 node_din <= n_r(63 downto 0);
                end case;
                node_absorb <= '1';
                counter <= counter+1;
                st<=st0;
125           else
                st <= st1;
                node_absorb <= '0';
                node_go <= '1';
              end if;
130
            when st1 =>
              node_go <= '0';
              st <= st1a;

135         when st1a =>
```

```
            if(node_ready='0') then
              st <= st1;
            else
              st <= END_HASH1;
            end if;

          when END_HASH1 =>
            if(node_ready='1') then
              node_squeeze <= '1';
              st <= END_HASH2;
              counter <= 0;
            end if;

          when END_HASH2 =>
            node_squeeze <= '1';
            case counter is
              when 0 =>
                node_out(255 downto 192) <= node_dout;
              when 1 =>
                node_out(191 downto 128) <= node_dout;
              when 2 =>
                node_out(127 downto 64) <= node_dout;
              when others =>
                node_out(63 downto 0) <= node_dout;
            end case;

            if(counter<3) then
              counter <= counter+1;
            else
              node_squeeze <= '0';
              counter <= 0;
              st <= stop_k;
              s_node_done <= '1';
            end if;

          when stop_k =>
            s_node_done <= '0';
            st <= initial;

          end case;
        end if;
      end process;

  end rtl;
```

B merkle_pkg.vhd

```
library ieee;
use ieee.std_logic_1164.all;

package merkle_pkg is
  type leaves_t is array(natural range <>) of
  std_logic_vector(255 downto 0);
  type wit_type is array(natural range <>) of
  std_logic_vector(256*3-1 downto 0);
end;
```

C merkle_tree.vhd

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

library work;
use work.merkle_pkg.all;

entity merkle_tree is
  generic(
    leaves_nbr  : integer range 0 to 16 := 8;
    logl        : integer range 0 to 4 := 3
  );
  port(
    clk         : in  std_logic;
    rst_n       : in  std_logic;
    tree_go     : in  std_logic;
    hash_out    : out std_logic_vector(255 downto 0)
    := (others => '0');
    leaves      : in  leaves_t(0 to leaves_nbr-1);
    hash_done   : out std_logic := '0';
    witness     : out wit_type(leaves_nbr-1 downto 0)
  );
end merkle_tree;

architecture rtl of merkle_tree is
  component merkle_node is
    port(
      clk         : in  std_logic;
      rst_n       : in  std_logic;
```

```
30        start_l      : in  std_logic;
          start_r      : in  std_logic;
          node_done    : out std_logic;
          n_l          : in  std_logic_vector(255 downto 0);
          n_r          : in  std_logic_vector(255 downto 0);
35        node_out     : out std_logic_vector(255 downto 0);

          ------------------------------
          -- keccak interface
          ------------------------------
40        k_init       : out std_logic;
          k_absorb     : out std_logic;
          k_go         : out std_logic;
          k_squeeze    : out std_logic;
          k_ready      : in  std_logic;
45        k_din        : out std_logic_vector(63 downto 0);
          k_dout       : in  std_logic_vector(63 downto 0);

          k_att        : in  std_logic
        );
50    end component merkle_node;

      component keccak16 is
      port (
        clk          : in  std_logic;
55      rst_n        : in  std_logic;
        init         : in  std_logic;
        go           : in  std_logic;
        absorb       : in  std_logic;
        squeeze      : in  std_logic;
60      din          : in  std_logic_vector(63 downto 0);
        ready        : out std_logic;
        dout         : out std_logic_vector(63 downto 0));
      end component;

65    type hash_array is array(1 to 2*leaves_nbr-1) of
      std_logic_vector(255 downto 0);
      signal s_hash : hash_array;

      signal s_done, s_init, s_go, s_absorb, s_squeeze,
70    s_ready, s_k_att :
      std_logic_vector(leaves_nbr-1 downto 1)
      := (others => '0');
      type data_array is array(leaves_nbr-1 downto 1) of
      std_logic_vector(63 downto 0);
75    signal s_din, s_dout : data_array;
```

```
    signal s_k_init, s_k_go, s_k_absorb, s_k_squeeze,
    s_k_ready :  std_logic;
    signal s_k_din, s_k_dout :
    std_logic_vector(63 downto 0);

    signal n_cntr : integer range 1 to leaves_nbr-1
    := leaves_nbr-1;

    signal r_hash_done : std_logic_vector(3 downto 0)
    := (others => '0');

  begin
    p_leaves_as_hash :
    for i in leaves_nbr to 2*leaves_nbr-1 generate
      s_hash(i) <= leaves(i-leaves_nbr);
    end generate;

    stages_loop : for y in 1 to logl-1 generate
      column : for x in 0 to 2**(y-1)-1 generate
        loop_node : merkle_node
          port map(
            clk => clk,
            rst_n => rst_n,
            start_l => s_done(2**y+2*x),
            start_r => s_done(2**y+2*x+1),
            node_done => s_done(2**(y-1)+x),
            n_l => s_hash(2**y+2*x),
            n_r => s_hash(2**y+2*x+1),
            node_out => s_hash(2**(y-1)+x),

            ------------------------------
            -- to or from keccak
            ------------------------------
            k_init => s_init(2**(y-1)+x),
            k_absorb => s_absorb(2**(y-1)+x),
            k_go => s_go(2**(y-1)+x),
            k_squeeze => s_squeeze(2**(y-1)+x),
            k_ready => s_ready(2**(y-1)+x),
            k_din => s_din(2**(y-1)+x),
            k_dout => s_dout(2**(y-1)+x),

            k_att => s_k_att(2**(y-1)+x)
          );
      end generate;
    end generate;
```

```
    last_stage : for x in 0 to leaves_nbr/2-1 generate
      last_line : merkle_node
125     port map(
          clk => clk,
          rst_n => rst_n,
          start_l => tree_go,
          start_r => tree_go,
130       node_done => s_done(leaves_nbr/2+x),
          n_l => leaves(2*x),
          n_r => leaves(2*x+1),
          node_out => s_hash(leaves_nbr/2+x),
          ------------------------
135       -- to or from keccak
          ------------------------
          k_init => s_init(leaves_nbr/2+x),
          k_absorb => s_absorb(leaves_nbr/2+x),
          k_go => s_go(leaves_nbr/2+x),
140       k_squeeze => s_squeeze(leaves_nbr/2+x),
          k_ready => s_ready(leaves_nbr/2+x),
          k_din => s_din(leaves_nbr/2+x),
          k_dout => s_dout(leaves_nbr/2+x),

145       k_att => s_k_att(leaves_nbr/2+x)
        );
    end generate;

    k_comp : keccak16
150   port map(
        clk          => clk,
        rst_n        => rst_n,
        init         => s_k_init,
        absorb       => s_k_absorb,
155     squeeze      => s_k_squeeze,
        go           => s_k_go,
        din          => s_k_din,
        ready        => s_k_ready,
        dout         => s_k_dout
160   );

    s_k_init <= s_init(n_cntr);
    s_k_absorb <= s_absorb(n_cntr);
    s_k_squeeze <= s_squeeze(n_cntr);
165 s_k_go <= s_go(n_cntr);
    s_ready(n_cntr) <= s_k_ready;
    s_k_din <= s_din(n_cntr);
```

```
    s_dout(n_cntr) <= s_k_dout;

p_out: process(clk, rst_n)
begin
  if rst_n = '0' then
    hash_out <= (others => '0');
    r_hash_done <= (others => '0');
    hash_done <= '0';
  elsif clk'event and clk = '1' then
    hash_out <= s_hash(1);
    r_hash_done(0) <= s_done(1);
    r_hash_done(3 downto 1) <= r_hash_done(2 downto 0);
    hash_done <= r_hash_done(3) or r_hash_done(2) or
    r_hash_done(1) or r_hash_done(0);
  end if;
end process;

tree_main: process(clk, rst_n, n_cntr)
begin
  if rst_n = '0' then
    n_cntr <= leaves_nbr-1;
  elsif clk'event and clk = '1' then
    if tree_go = '1' then
      n_cntr <= leaves_nbr-1;
    else
      if s_done(n_cntr)='1' then
        if n_cntr>1 then
          n_cntr <= n_cntr - 1;
        end if;
      end if;
    end if;
  end if;

  k_attr : for k in 1 to leaves_nbr-1 loop
    if k=n_cntr then
      s_k_att(k) <= '1';
    else
      s_k_att(k) <= '0';
    end if;
  end loop;
end process;

wit_process: process(rst_n, r_hash_done, s_hash)
  variable w_addr : std_logic_vector(logl downto 0)
  := (others => '0');
  variable ind : std_logic_vector(logl downto 0)
```

```
      := (others => '0');
  begin
    if rst_n = '0' then
      witness <= (others => (others => '0'));
      w_addr := (others => '0');
      ind := (others => '0');
    elsif r_hash_done(0)='1' then
      w_addr := (others => '0');
      for w in 0 to leaves_nbr-1 loop
        ind := std_logic_vector(to_unsigned(w+leaves_nbr,
        ind'length));
        for y in 2 to logl+1 loop
          for i in logl downto y loop
            w_addr(i) := '0';
          end loop;
          for i in y-1 downto 1 loop
            w_addr(i) := ind(i+logl+1-y);
          end loop;
          w_addr(0) := not(ind(logl+1-y));
          witness(w)(256*(y-1)-1 downto 256*(y-2)) <=
          s_hash(to_integer(unsigned(w_addr)));
        end loop;
      end loop;
    end if;
  end process;

end rtl;
```