

VHDL - Circuits et simulation

Exercice 1

Ecrire une entité qui nous permettra d'utiliser un registre (bascule D):

```
entity bascule is
  port(
    clk      : in  std_logic;
    input    : in  std_logic;
    output   : out std_logic
  );
end entity;
```

Il faut ensuite compiler le fichier `bascule.vhd` à l'aide de la commande bash:

```
ghdl -a bascule.vhd
```

Exercice 2

Ecrire une architecture qui inclut effectivement une bascule D:

```
architecture using_rising_edge of bascule is
begin

  process(clk)
  begin
    if rising_edge(clk) then
      output <= input;
    end if;
  end process;

end using_rising_edge;
```

Il faut ensuite compiler le fichier `bascule.vhd` à l'aide de la commande bash :

```
ghdl -a bascule.vhd
```

Exercice 3

Ecrire un testbench qui permet de tester cette bascule:

Pour générer automatiquement le "squelette" du test bench nous pouvons utiliser le gem ruby **vhdl_td**:

```
ghdl -a bascule.vhd
vhd1_tb bascule.vhd
```

Après cela, un fichier `bascule_tb.vhd` a été créé, il réutilise des propriétés de l'entité et de l'architecture de notre bascule D.

Dans ce testbench il faut ensuite créer un stimuli:

```
stim : process
begin
    report "running testbench for dff(using_rising_edge)";
    report "waiting for asynchronous reset";
    wait until reset_n='1'; --attendre un "1"
    wait_cycles(1); --attendre 1 cycle
    report "applying stimuli...";
    input <= '1'; --input = 1
    wait_cycles(1); --attendre 1 cycle
    report "applying stimuli...";
    input <= '0'; --input = 0
    wait_cycles(1);
    report "applying stimuli...";
    input <= '1';
    wait_cycles(4);
    input <= '0';
    wait_cycles(5);
    report "applying stimuli...";
    input <= '1';
    wait_cycles(2);
    report "end of simulation";
    running <= false;
    wait;
end process;
```

Afin de visualiser le signal, nous allons utiliser le wave viewer `GTKWave`

Afin d'automatiser la suite de commande nous allons créer un script. Voici les commandes que nous allons automatiser:

```
ghdl -a bascule.vhd
echo 'VHDL compiled, Binary generated!'
echo '===== '

ghdl -a bascule_tb.vhd
echo 'Testbench compiled, Binary generated!'
echo '===== '

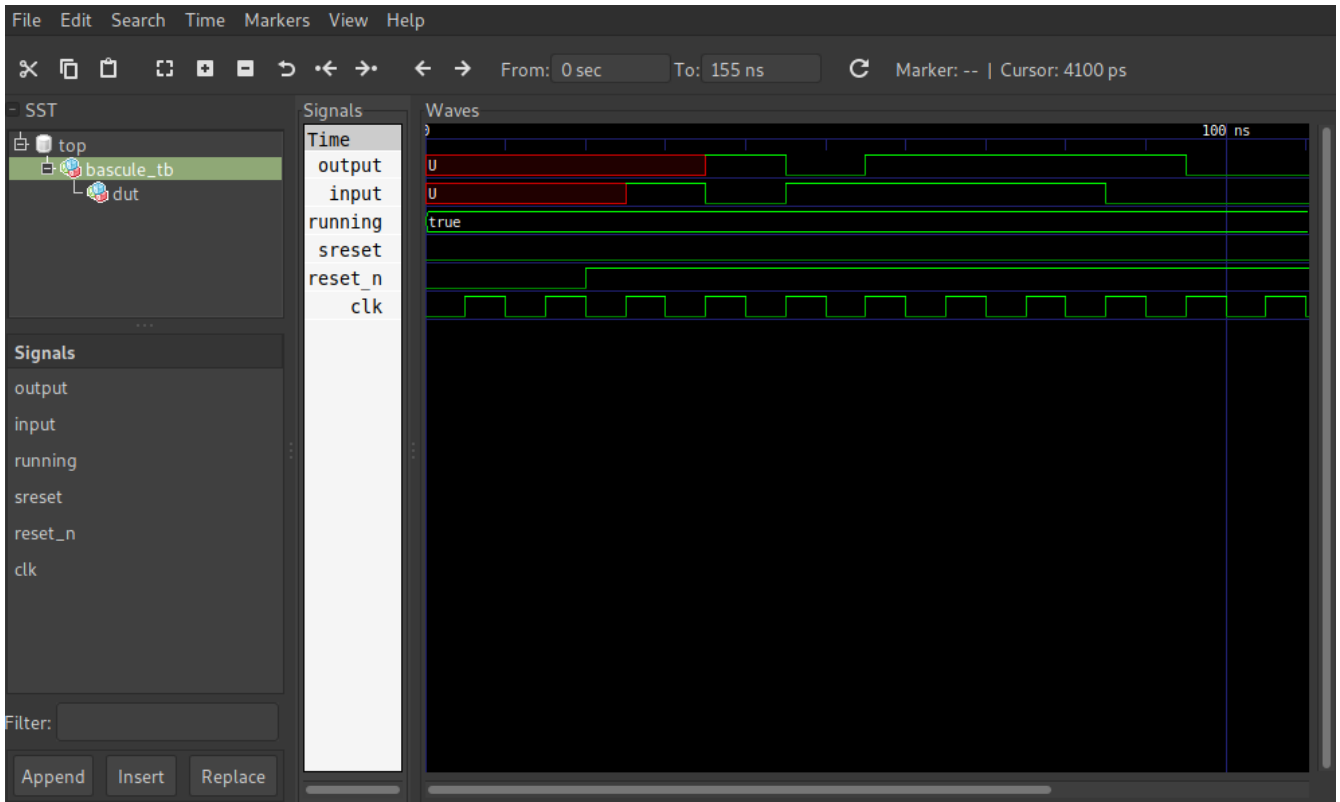
ghdl -e bascule_tb
echo 'Elaboration completed!'
echo '===== '

ghdl -r bascule_tb --wave=bascule.ghw
echo 'Waveform generated!'
```

```
echo '====='
```

```
gtkwave bascule.ghw
```

Aperçu du fichier `bascule.ghw` avec le viewer GTKWave:



Exercice 4

Dupliquer et modifier l'entité et l'architecture de manière à rendre ce registre générique, sur N bits:

Voici l'entité et l'architecture d'une bascule D générique:

```
entity bascule_gen is
    generic ( N : natural := 4);
    port (
        clk,reset  : in std_logic;
        ENTREE     : in std_logic_vector (N-1 downto 0);
        SORTIE     : out std_logic_vector (N-1 downto 0)
    );
end entity;

architecture rtl of bascule_gen is

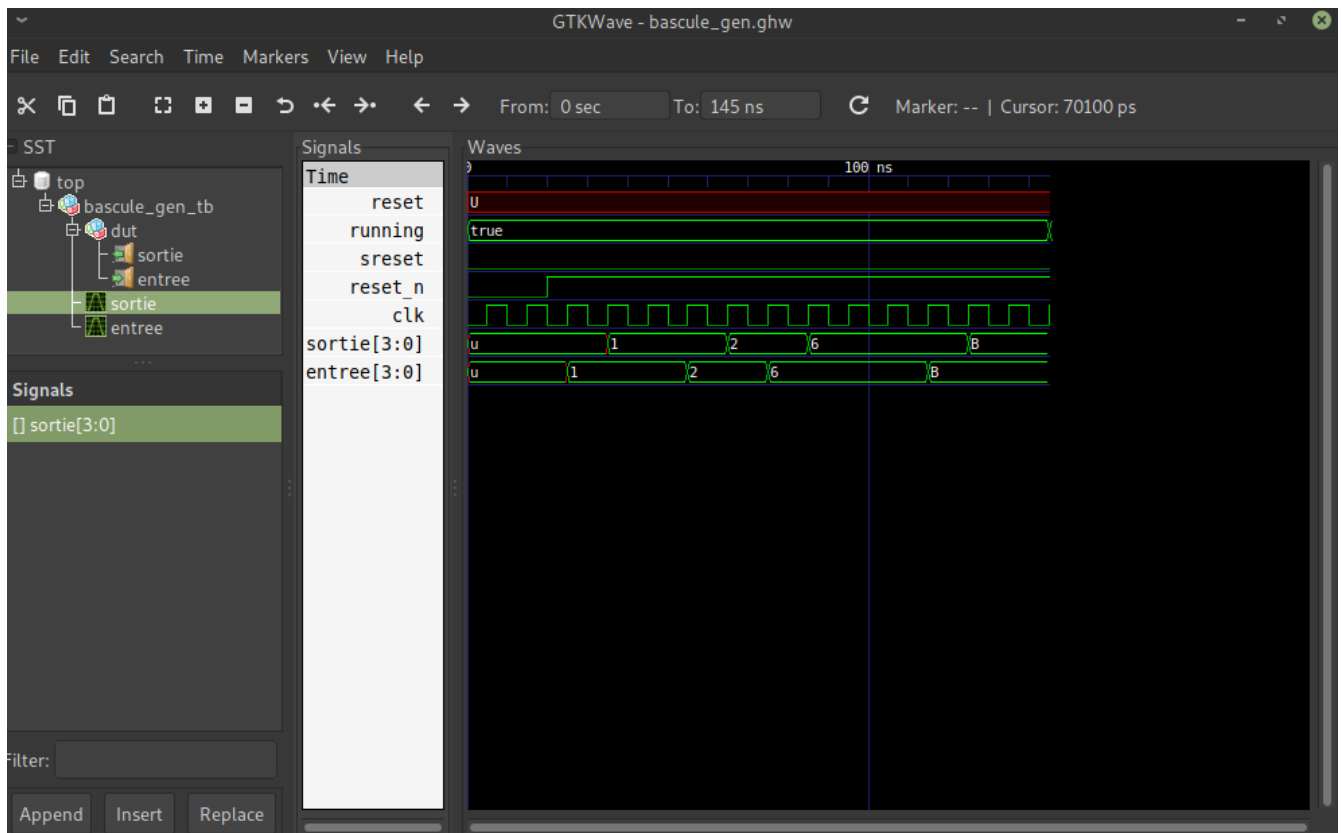
begin --
    P1:process(clk,reset)
    begin
        if reset='0' then
            SORTIE <= (others => '0');
        elsif rising_edge(clk) then
            SORTIE <= ENTREE;
        end if;
    end process;
end architecture;
```

```
end process;  
end rtl;
```

Nous créons ensuite un testbench:

```
stim : process  
begin  
    report "running testbench for dff(using_rising_edge)";  
    report "waiting for asynchronous reset";  
    wait until reset_n='1';  
    wait_cycles(1);  
    report "applying stimuli...";  
    ENTREE <= "0001";  
    wait_cycles(3);  
    report "applying stimuli...";  
    ENTREE <= "0010";  
    wait_cycles(2);  
    report "applying stimuli...";  
    ENTREE <= "0110";  
    wait_cycles(4);  
    report "applying stimuli...";  
    ENTREE <= "1011";  
    wait_cycles(3);  
    report "end of simulation";  
    running <= false;  
    wait;  
end process;
```

GTKWave signal:



Exercice 5

Créer un compteur qui permet de compter, lorsqu'on l'autorise, jusqu'à 1000 et qui se remet automatiquement à 0.
Tester:

Nous définissons l'entité et l'architecture de notre compteur:

```
entity Compteur is
  port (
    reset_n : in std_logic;
    clk      : in std_logic;
    qs       : out std_logic_vector(9 downto 0));
end Compteur;

architecture bhv of Compteur is
  signal q : unsigned(9 downto 0);
begin

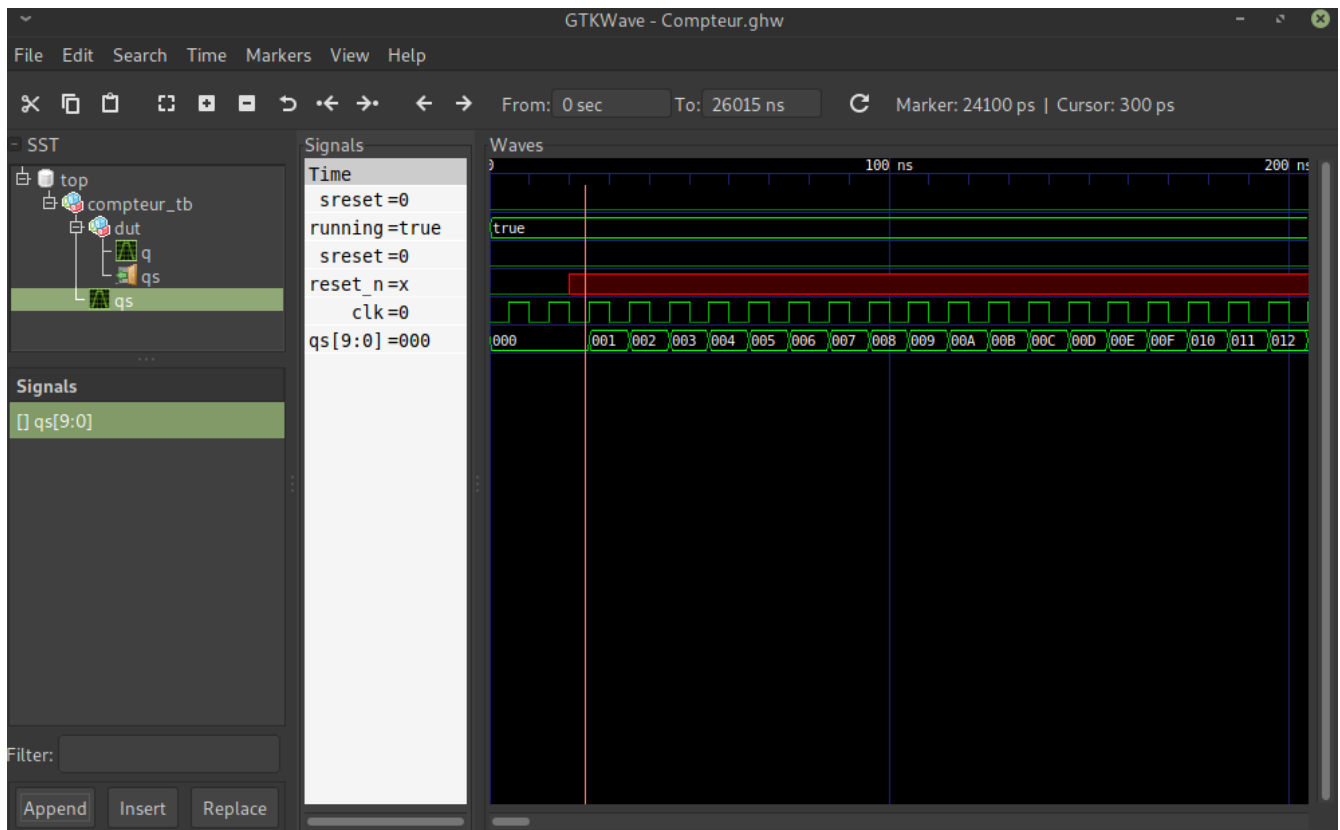
  process(clk)
  begin
    if reset_n = '0' then
      q <= (others => '0');
    elsif rising_edge(clk) then
      q <= q + 1;
      if q = 1000 then -- si le compteur arrive à 1000
        q <= (others => '0'); -- back to zero !
      end if;
    end if;
  end process;
end architecture;
```

```
    end if;  
end process;  
  
qs <= std_logic_vector(q);  
  
end bhv;
```

Nous créons ensuite un testbench:

```
stim : process  
begin  
    reset_n <= '0';--inactif  
  
    for i in 0 to 100 loop  
        wait until rising_edge(clk);  
    end loop;  
  
    reset_n <= '1';--actif  
    report "now active!";  
  
    for i in 0 to 2500 loop  
        wait until rising_edge(clk);  
    end loop;  
    running <= false;  
    wait;  
end process;
```

GTKWave signal:



Exercice 6

Créer une architecture qui permet, au choix d'un utilisateur, de réaliser une addition, une multiplication, une soustraction ou l'une des opérations logiques bits-à-dit entre 2 données signées soumises par l'utilisateur.

Ce type de circuit s'appelle une UAL (ALU) : unité arithmétique et logique, au cœur du cœur de n'importe quel ordinateur:

Entité et architecture de notre UAL:

```
entity ual is
  port(
    a, b : in  unsigned(3 downto 0);
    op  : in  std_logic_vector(2 downto 0);
    res : out unsigned(3 downto 0)
  );
end entity;

architecture rtl of ual is
begin

  ual_proc : process(a, b, op)
  begin
    res <= (others => '0'); --all bits to '0'
    case op is
      when "000" =>    --+
        res <= resize(a + b,4);
      when "001" =>    ---
```

```

        res <= resize(a - b,4);
    when "010" =>    --*
        res <= resize(a * b,4);
    when "011" =>    --and
        res <= a and b;
    when "100" =>    --or
        res <= a or b;
    when "101" =>    --xor
        res <= a xor b;
    when "111" =>    --not
        res <= not(a);
    when others =>    --all other operations have no effect
        null;
    end case;
end process;

end rtl;

```

Nous créons ensuite un testbench:

```

stim : process
begin
    report "running testbench for ual(rtl)";
    report "waiting for asynchronous reset";
    wait until reset_n='1';
    wait_cycles(100);
    report "applying stimuli...";
    a <= "0111"; -- a=7
    b <= "0011"; -- b=3
    wait_cycles(100);
    report "applying stimuli...";
    op <= "000"; --operation +
    wait_cycles(100);
    op <= "001"; --operation -
    wait_cycles(100);
    op <= "010"; --operation *
    wait_cycles(100);
    op <= "011"; --operation and
    wait_cycles(100);
    op <= "100"; --operation or
    wait_cycles(100);
    op <= "101"; --operation xor
    wait_cycles(100);
    op <= "111"; --operation not
    wait_cycles(100);
    a <= "0101"; -- a=5
    b <= "0101"; -- b=5
    wait_cycles(100);
    report "applying stimuli...";
    op <= "000";
    wait_cycles(100);
end process;

```

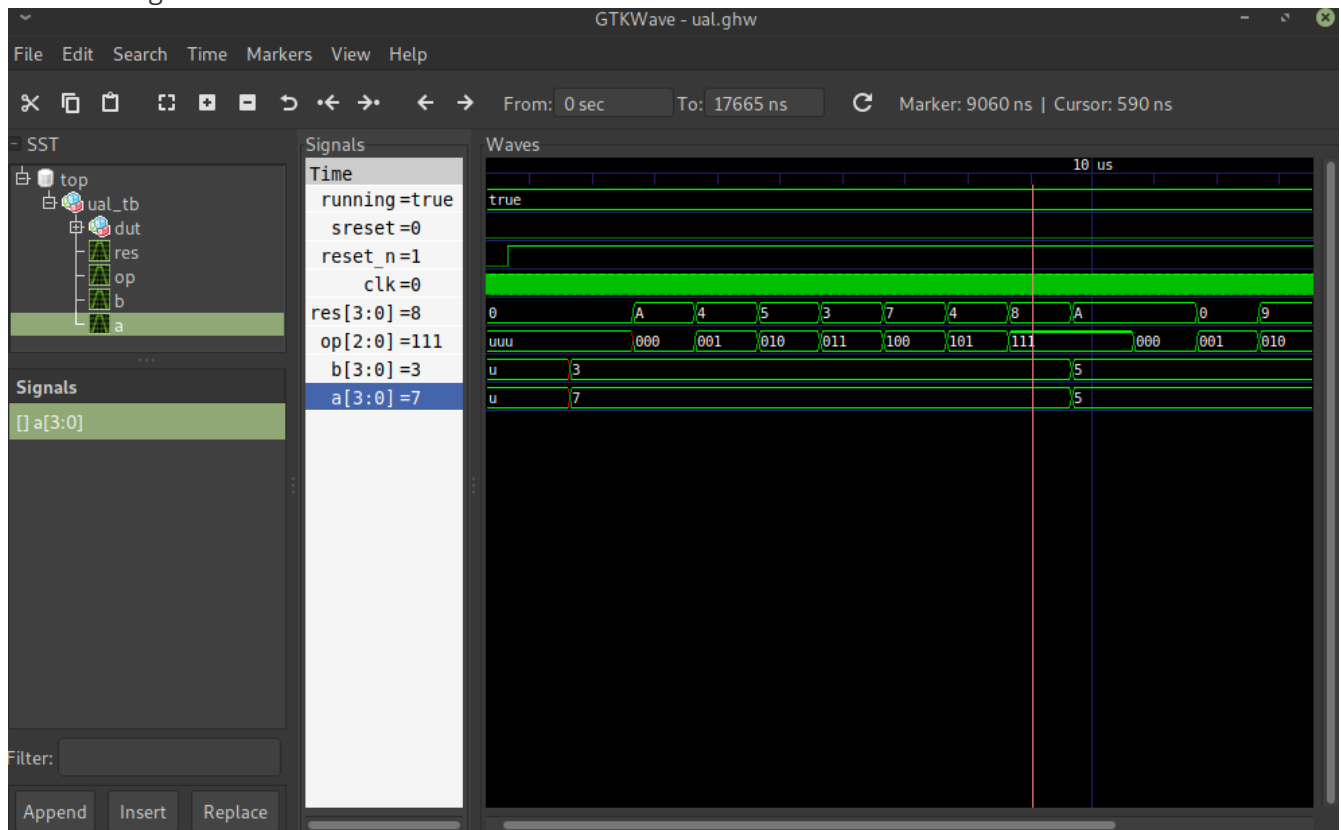


```

op <= "001";
wait_cycles(100);
op <= "010";
wait_cycles(100);
op <= "011";
wait_cycles(100);
op <= "100";
wait_cycles(100);
op <= "101";
wait_cycles(100);
op <= "111";
wait_cycles(100);
report "end of simulation";
running <= false;
wait;
end process;

```

GTKWave signal:



Attention, il ne permet pas l'opération d'entiers signés!

Exercice 7

Créer une mémoire adressable et synthétisable pour des octets, pouvant stocker 128 octets. Réaliser un banc de test qui permet de lire et écrire dans cette mémoire:

Entité et architecture de notre RAM:

```
entity ram is
  port(
    reset_n : in  std_logic;
    clk      : in  std_logic;
    wr       : in  std_logic;
    address  : in  unsigned(7 downto 0);
    datain   : in  std_logic_vector(7 downto 0);
    dataout  : out std_logic_vector(7 downto 0)
  );
end entity;

architecture rtl of ram is
  type memory_type is array(0 to 127) of std_logic_vector(7 downto 0);
  signal mem       : memory_type;
  signal addr_r    : unsigned(7 downto 0);
begin

  ram_proc : process(reset_n,clk)
  begin

    if reset_n='0' then
      for i in 0 to 127 loop
        mem(i) <= (others => '0');
      end loop;
      addr_r <= to_unsigned(0,8);

    elsif rising_edge(clk) then
      if wr='1' then
        mem(to_integer(unsigned(address))) <= datain;
      end if;
      addr_r <= address;
    end if;
  end process;

  dataout <= mem(to_integer(addr_r));

end rtl;
```

Testbench:

```
stim : process
  begin
    report "running testbench for ram(rtl)";
    report "waiting for asynchronous reset";
    wait until reset_n='1';
    wait_cycles(100);

    -- Test WR=1 pour autorisation ecriture
    wr <= '1';
```

```

wait_cycles(10);
--écriture de 4 à l'adresse 2
    address <= to_unsigned(2,8);
    datain <= std_logic_vector(to_unsigned(4,8));
    wait_cycles(10);
wr <='0';
-- fin d'autorisation d'écriture
wait_cycles(10);
-- Test WR=1 pour autorisation ecriture
    wr <='1';
wait_cycles(10);
--écriture de 5 à l'adresse 3
    address <= to_unsigned(3,8);
    datain <= std_logic_vector(to_unsigned(5,8));
wait_cycles(10);
wr <='0';
-- fin d'autorisation d'écriture
wait_cycles(10);
-- Test WR=1 pour autorisation ecriture
    wr <='1';
wait_cycles(10);
--écriture de 7 à l'adresse 6
    address <= to_unsigned(6,8);
    datain <= std_logic_vector(to_unsigned(7,8));
    wait_cycles(10);
wr <='0';
-- fin d'autorisation d'écriture
wait_cycles(10);

-- lecture de l'adresse 2
    wait_cycles(10);
    address <= to_unsigned(2,8);
-- lecture de l'adresse 3
    wait_cycles(10);
    address <= to_unsigned(3,8);
-- lecture de l'adresse 6
    wait_cycles(10);
    address <= to_unsigned(6,8);

```

GTKWave signal:

