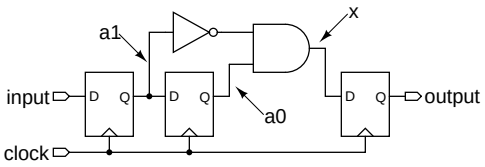
Assignment 1 Solutions

Question 1

(a) We can label the signals in the schematic as follows:



and write the following VHDL to implement it:

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

entity sol1q1 is
  port (
    input, clock : in  std_logic;
    output : out std_logic);
end sol1q1;

architecture rtl of sol1q1 is
  signal a0, a1, x : std_logic;
begin

  x <= not a1 and a0 ;
  -- or:
  -- x <= '1' when a1='0' and a0='1' else '0' ;

  process (clock)
  begin  -- process
    if clock'event and clock = '1' then
      a1 <= input ;
      a0 <= a1 ;
      output <= x ;
    end if;
  end process;

end rtl;
```

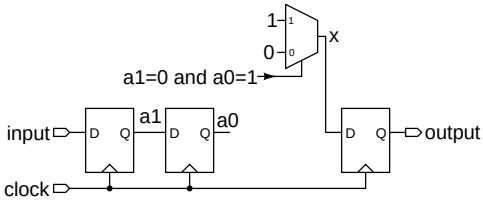
which gives the simulation results shown on the last page.

Note that the order of signal assignments in the process doesn't matter – all take effect simultaneously at the end of the process.

(b) We can replace the assignment statement with the conditional assignment:

```
x <= '1' when a1='0' and a0 = '1' else '0' ;
```

and since a conditional assignment corresponds to a multiplexer we can draw the schematic as follows:



(c) This circuit detects a change in the input from high to low, in other words, a falling edge.

The input is "registered" which adds a delay of between 0 and one clock cycle (it's a half cycle in the simulation above). The output is also registered which results in an additional delay of one clock cycle in detecting the falling edge.

Note that if I had used an `xor` gate instead of an inverter and `and` gate (as hinted at in the question), the circuit would be able to detect both rising and falling edges.

Question 2

(a) The design of a state machine to determine the direction of motion is done using the 3-step approach described in the Lecture 2 notes:

Step 1: Determine inputs and outputs. As stated in the question, the inputs are the quadrature encoder outputs `A` and `B`, and the output is `right` indicating motion toward the right.

Step 2: Determine state variables. Registered outputs should be used as state variables and therefore `right` will be a state variable. It will have possible values 0 (moving left) and 1 (moving right).

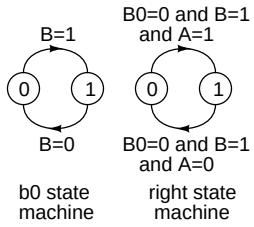
It doesn't seem possible to determine the direction of motion given only the previous direction (the value of `right`) and the two inputs (A and B) so we'll add additional state variables.

Based on the question, the behaviour of the state machine depends on the history of the inputs (i.e., which input changed first). To detect changes in an input we need to have access to the previous value as well as the current value. We can save the previous values of either input using flip-flops as shown in the first question.

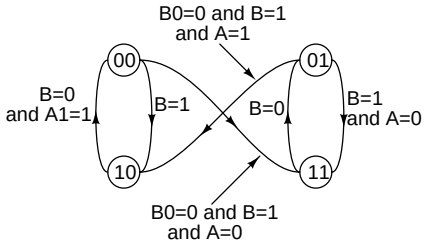
Step 3: State transition conditions. According to the problem description, the `right` output (state) should be set true if "A goes high before B." We can do this by testing A when B goes high. If A is already high when B goes high then A went high (changed) before B and we are moving right. We can determine when B goes high by comparing the current and previous values of B. It therefore appears we might only need one additional state variable, the previous value of B which we'll label `b0`.

The state transition condition is simply that when there is a rising edge on B (b0=0 and B=1) we set `right=A`.

It's easiest to draw this as two separate state machines, one for detecting rising edges on B and one for changing the `right` output:



although we could combine the two into one:



note: MS bit is B0, LS bit is right

(b) The corresponding VHDL is shown below. Unfortunately, `right` is a reserved word in VHD so I've used `right_out` as the output signal name. I've used another signal, `output`, in the architecture since the value of an `output` port signal cannot be used inside the architecture.

It is good practice to register all inputs (for reasons that will be explained later). In this case we would have to use flip-flops on both the A and B inputs. The solution where both inputs are registered is shown commented out below.

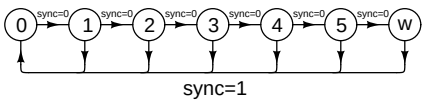
The VHDL code and simulation results are given on the last page.

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

entity sol1q2 is
  port (
    a, b, clock : in std_logic;
    right_out : out std_logic);
end sol1q2;

architecture rtl of sol1q2 is
  signal b0, output, next_output : std_logic;
  -- signal a1, b0, b1, output, next_output : std_logic;
begin

  next_output <= a when b0 = '0' and b = '1' else output ;
  -- next_output <= a1 when b0 = '0' and b1 = '1' else output ;

  process (clock)
  begin
    if clock'event and clock='1' then
      b0 <= b ;
      -- a1 <= a ;
      -- b1 <= b ;
      -- b0 <= b1;
      output <= next_output ;
    end if;
  end process;

  right_out <= output ;

end rtl;
```

We could improve the resolution of the quadrature encoder by considering both rising and falling edges of both signals but this is not required by the question.

Question 3

(a) The state machine controls which bit of the parallel output is to be loaded with the serial input. This means there must be at least 6 states where we store the input bit (labelled 0 to 5 below) and a state where all bits have been filled and none of the bits should be changed. This state is labelled 'w'.



(b) Three possible solutions are shown as three architectures for the same entity.

The solution in architecture `rtl1` uses a conditional assignment for each output bit to update that bit only in the appropriate state. This solution could be made much less verbose if we had used a VHDL feature (`for...generate`) that we have not covered.

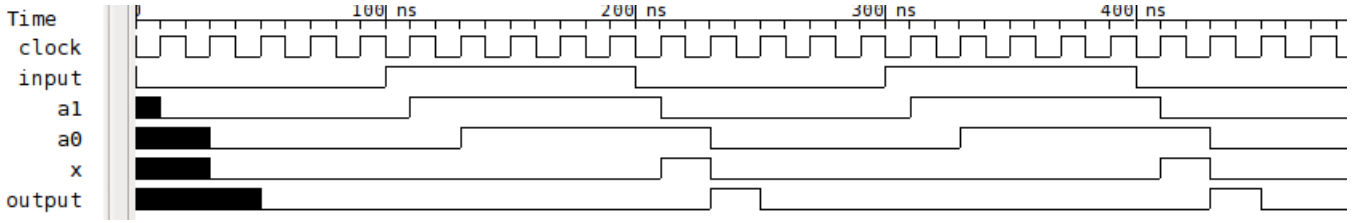
The second solution uses a 7x1-bit array to model a RAM and an `unsigned` state variable that specifies the RAM address to be written. To simplify the design we use a 7-bit RAM and write all bits following the 6'th bit to an (unused) bit 6.

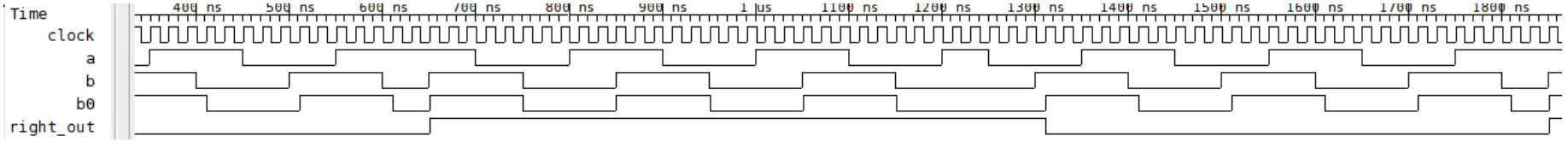
The third solution uses a bit mask approach. The state variable is a 6-bit `std_logic_vector` variable where each state has a '1' only in the position that is to be filled with the serial input data. The state is reset to "000001" and shifted left on each clock. After six clock edges the value will be zero and no more input bits will be stored.

Many other solutions are possible.

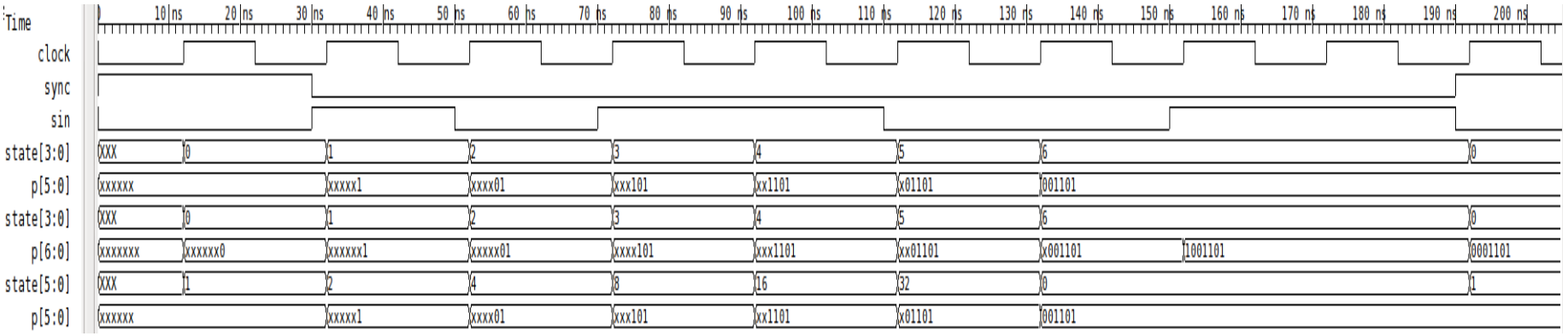
The VHDL code and the simulation results for the three architectures is given below.

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

entity sol1q3 is
  port (
    sin, sync, clock : in  std_logic;
    pout             : out std_logic_vector (5 downto 0));
end sol1q3 ;

-- solution using individual conditional assignments

architecture rtl1 of sol1q3 is

  signal p, next_p : std_logic_vector (5 downto 0) ;
  signal state, next_state : unsigned (3 downto 0);

begin

  next_state <=
    to_unsigned(0,state'length) when sync = '1' else
    state + 1 when state /= 6 else
    state ;

  next_p(0) <= sin when state = 0 else p(0) ;
  next_p(1) <= sin when state = 1 else p(1) ;
  next_p(2) <= sin when state = 2 else p(2) ;
  next_p(3) <= sin when state = 3 else p(3) ;
  next_p(4) <= sin when state = 4 else p(4) ;
  next_p(5) <= sin when state = 5 else p(5) ;

  process(clock)
  begin
     if clock'event and clock='1' then
         state <= next_state ;
         p <= next_p ;
     end if ;
  end process ;

  pout <= p ;

end rtl1;

-- solution using an array (7x1 RAM)

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

architecture rtl2 of sol1q3 is

  signal p : std_logic_vector (6 downto 0) ;
  signal state, next_state : unsigned (3 downto 0);

begin

  next_state <=
    to_unsigned(0,state'length) when sync = '1' else
    state+1 when state /= 6 else
    state ;

  process(clock)
  begin
     if clock'event and clock='1' then
         state <= next_state ;
         p(to_integer(state)) <= sin ;
     end if ;
  end process ;

  pout <= p(5 downto 0) ;

end rtl2;

-- solution using a bit mask

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

architecture rtl3 of sol1q3 is

  signal p, next_p, state, next_state :
    std_logic_vector (5 downto 0) ;

begin

  next_state <=
    "000001" when sync = '1' else
    state(4 downto 0) & "0" when state /= "000000" else
    state ;

  next_p <=
    ( p or state ) when sin = '1' else
    ( p and not state ) ;

  process(clock)
  begin
     if clock'event and clock='1' then
         state <= next_state ;
         p <= next_p ;
     end if ;
  end process ;

  pout <= p ;

end rtl3;
```


Time
clock
input
a1
a0
x
output
0
100 ns
200 ns
300 ns
400 ns


Time
clock
a
b
b0
right_out
400 ns
500 ns
600 ns
700 ns
800 ns
900 ns
1 us
1100 ns
1200 ns
1300 ns
1400 ns
1500 ns
1600 ns
1700 ns
1800 ns

4


Time
clock
sync
sin
state[3:0]
p[5:0]
state[3:0]
p[6:0]
state[5:0]
p[5:0]
0
10 ns
20 ns
30 ns
40 ns
50 ns
60 ns
70 ns
80 ns
90 ns
100 ns
110 ns
120 ns
130 ns
140 ns
150 ns
160 ns
170 ns
180 ns
190 ns
200 ns
XXX
xxxxxx
xxxxxx1
xxxx01
xxx101
xx1101
x01101
001101
xxxxxxx
xxxxxx0
xxxxxx1
xxxxx01
xxxx101
xxx1101
xx01101
x001101
1001101
0001101
1
2
4
8
16
32