Digital Systems exam (2 hours)

Renaud Pacalet - 2026-06-11

The text in black is the original one.

The text in red is examples of the expected correct answers. Only this text was expected, possibly in shorter form, nothing more.

The text in blue is extra comments about the expected correct answers.

Warning: the course changes frequently (content, vocabulary, examples…); some questions and answer proposals can thus be partly or completely out of scope. Warning: some questions can be answered in many different ways; the proposed answers are just examples and they are not exhaustive.

The exam is closed book, closed notes, except one A4-sized, double sided, cheat sheet. Calculators are allowed but communicating devices are strictly forbidden. Please number the different pages of your paper and indicate on each page your first and last names. You can write your answers in French or in English, as you wish. Precede your answers with the question's number. If some information or hypotheses are missing to answer a question, add them. If you consider a question as absurd and thus decide to not answer, explain why. If you do not have time to answer a question but know how to, briefly explain your ideas. Advice: quickly go through the document and answer the easy parts first. The first question is worth 4 points. The other questions are worth 2 points each. The problem is worth 10 points Use the provided VHDL cheat sheets at the end of this paper if you don't remember the syntax.

1. Questions

1.1. VHDL coding of a DFF with asynchronous reset

Bob has been asked to code in VHDL a D-flip-flop synchronized on rising edges of its clock and with active low **asynchronous** reset. The expected behaviour is represented by the waveforms of Figure 1 where `clk` is the clock, `rstn` is the reset, `d` is the data input and `q` is the data output.

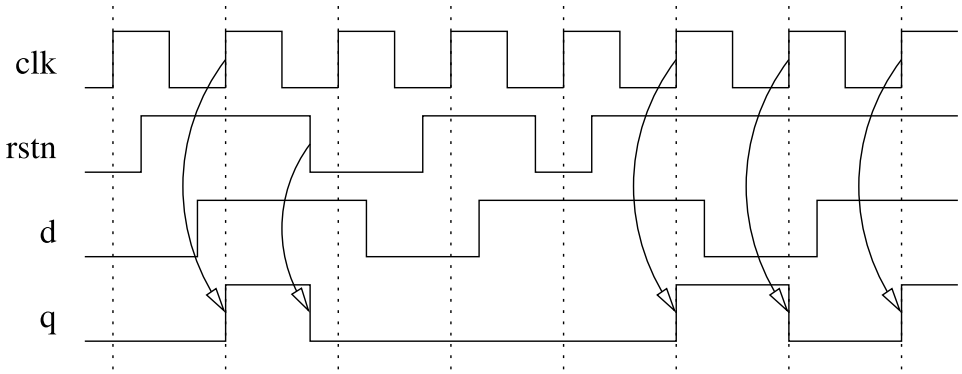

Figure 1: The expected behaviour of the DFF

1.1.1. Bob's first coding attempt

Bob's first coding attempt is shown on Listing 1. Bob also coded a simulation environment for his DFF, that reproduces the behaviour of `clk`, `rstn` and `d` of Figure 1. He compiles with `ghdl`, without errors, and tries to simulate, but then `ghdl` raises an error:

```
$ ghdl -a --std=08 dff.vhd dff_sim.vhd
$ ghdl -r --std=08 dff_sim
dff.vhd:6:8:error: too many drivers for port "q"
       q:            out std_ulogic);
       ^
```

Can you explain this error? Where does it come from?

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

entity dff is
  port(clk, rstn, d: in  std_ulogic;
       q:            out std_ulogic);
end entity dff;

architecture rtl of dff is
begin

  reset: process(rstn)
  begin
    if rstn = '0' then
      q <= '0';
    end if;
  end process;

  clock: process(clk)
  begin
    if rising_edge(clk) then
      q <= d;
    end if;
  end process;

end architecture rtl;
```

Listing 1: Bob's first coding attempt

The `q` port is of type `std_ulogic`, which is unresolved, so `q` cannot be driven by two or more processes, as Bob tried to do.

> Only signals of resolved types, like `sdt_logic`, can be driven by several processes. These types have an associated resolution function that is called by the simulator to compute the resulting value from the different values that the different processes drive.

1.1.2. Bob's second coding attempt

Bob asks how to fix this to the new AI of the company, which suggests to replace type `std_ulogic` with type `std_logic`. Bob edits his two VHDL source files, re-compiles and simulates. Unfortunately, the waveforms (see Figure 2) are not what Bob was expecting

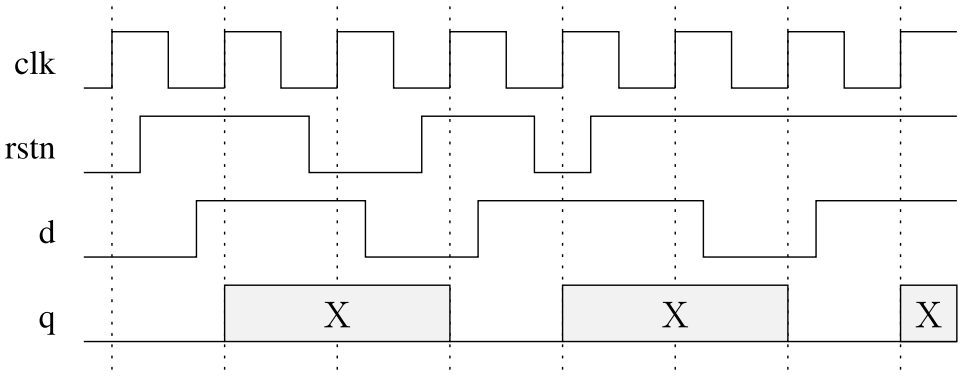

Figure 2: The behaviour of Bob's second attempt

Can you explain this behaviour? Where does it come from? What do you think of the AI's suggestion?

As the `q` port is now of type `std_logic`, which is resolved, there is no error, even if there are two driving processes. On the first value change to `'0'` of `rstn` the first process (`reset`) starts driving `q` to value `'0'`, and this driving never stops until the end of the simulation, even when `rstn` changes again. As long as the two processes agree on the driving value everything is fine, but when they disagree, and one drives `'0'` while the other drives `'1'`, the resolution function outputs `'X'` (unknown). The AI's suggestion is indeed a way to fix the error message that Bob reported, but it does not help implementing the expected behaviour.

1.1.3. Bob's third coding attempt

Bob decides to ask Mary, who seems to be better at VHDL coding than the new AI. Mary recommends to use type `std_ulogic` but only one process instead of two. Bob reworks his code as shown on Listing 2. He compiles and simulates. The waveforms he obtains are shown on Figure 3.

```
1  architecture rtl3 of dff is
2  begin
3
4    process(clk)
5    begin
6      if rstn = '0' then
7        q <= '0';
8      elsif rising_edge(clk) then
9        q <= d;
10     end if;
11   end process;
12
13 end architecture rtl3;
```

Listing 2: Bob's third coding attempt

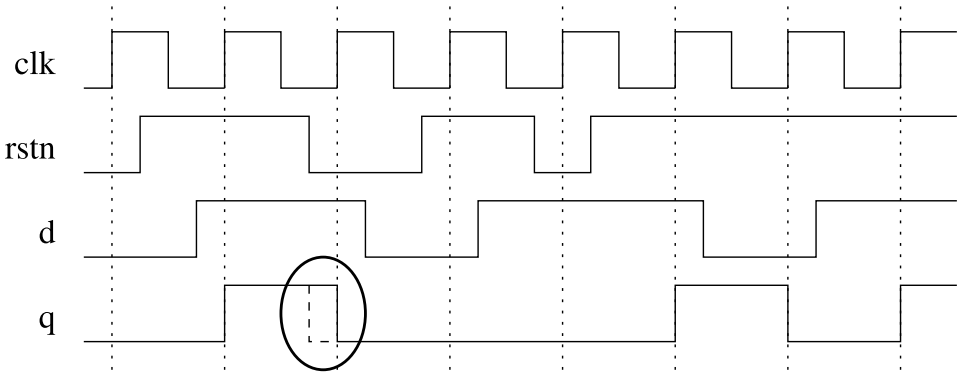

Figure 3: The behaviour of Bob's second attempt

Bob is very happy because this time everything looks fine, until he shows that to Jane, his manager, who immediately spots a difference with the expected behaviour. On Figure 3 the expected behaviour of output `q` is represented as a dashed line and the difference is circled.

Can you explain this behaviour? Where does it come from?

The reset is supposedly **asynchronous** but as Bob forgot to add it in the sensitivity list of his process, it is taken into account only on value changes of `clk` (rising **and** falling edges), which is not the expected behaviour. This is why the first falling edge of `rstn` does not have an immediate effect on `q`; it is only on the following value change of `clk` that `q` is assigned `'0'`.

1.1.4. Your coding attempt

Bob is a bit desperate and decides to ask your help, because, even if you are very busy, he knows that you will solve this in a matter of minutes. Write down a correct architecture for the DFF, explain the differences with Bob's third attempt.

Simply adding the reset `rstn` to the sensitivity list of the process solves the issue:

```
...
process(clk, rstn)
...
```

1.2. Range of integer variables and signals

Why is it preferable to specify the range of variables and signals of integer type?

First because a synthesizer uses this information to instantiate the exact bit number: an integer type object without range constraint synthesizes in a 32 bits object. Second because the simulator dynamically checks the constraint and this is an interesting debugging feature.

1.3. State machines

What are the differences between a Mealy and a Moore state machine? What are their advantages and drawbacks?

In a Mealy machine the outputs depend on the current state and on the inputs. In a Moore state machine the outputs depend only on the current state.

In some cases, for the same functional specification, a Mealy state machine requires fewer states than its Moore counterpart, which may be considered as an advantage, especially when it leads to a smaller and less power consuming design.

The main drawback of Mealy machines is their combinational paths between the primary inputs and the primary outputs. When connecting Mealy machines in a chain, the critical path increases while it doesn't with Moore machines.

If the outputs of a state machine are control signals of a datapath (a frequent situation) they may be connected to many logic gates (e.g., multiplexers), and consenquently they can be heavily loaded (capacitive). In such situations they can be slow changing signals, which increases the critical path of the whole design. In such cases Moore could be preferred.

Note: when it is feasible, computing these control signals one clock cycle ahead and sampling them in registers can be an interesting solution. It reduces the impact on the critical path to its minimum and it makes Moore and Mealy equally usable.

1.4. Logic synthesis

In your opinion, how many bits of register will be inferred when synthesizing the VHDL code of Listing 3? Draw a schematic of the synthesis result. Could we obtain the same behaviour with less registers?

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  entity bsk is
5    port(clk:  in  std_ulogic;
6         din:  in  std_ulogic_vector(11 downto 0);
7         dout: out std_ulogic_vector(11 downto 0));
8  end entity bsk;
9
10 architecture rtl of bsk is
11   signal a, b: std_ulogic_vector(11 downto 0);
12 begin
13   process(clk)
14   begin
15     if rising_edge(clk) then
16       a    <= din;
17       b    <= a;
18       dout <= a or b;
19     end if;
20   end process;
21 end architecture rtl;
```

Listing 3: Synthesizable model

36 bits of register will be inferred.

The 2 signals `a`, `b` and the output port `dout` are assigned on rising edges of `clk`. So, one register is inferred by the synthesizer for each of them. They are 12 bits wide, so $3 \times 12 = 36$ bits of register are inferred.

The schematic is shown on Figure 4.

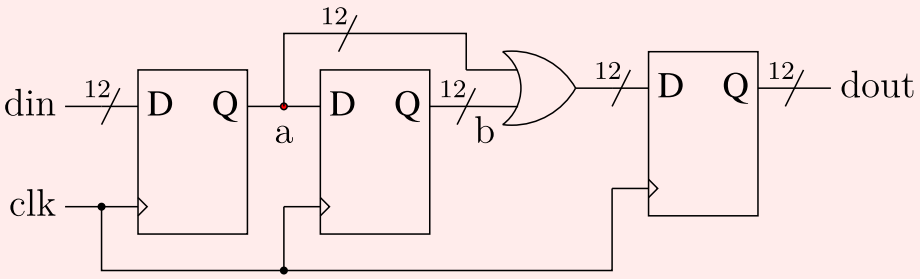

Figure 4: Schematic of the synthesis result

There is no way to obtain the **exact** same timing **and** functional behaviour with less registers.

But if the goal is only to compute the bitwise OR of two consecutive values of `din`, and the timing of the output is not relevant, a 12-bits register could suffice, as shown on Figure 5 and Listing 4.

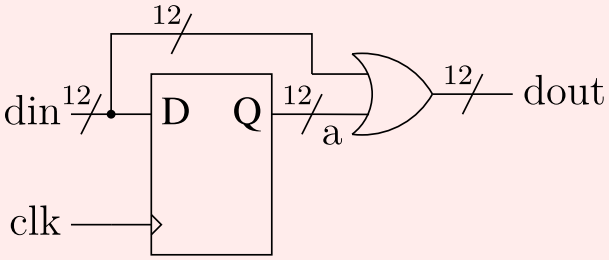

Figure 5: Schematic of the synthesis result of the optimized version

```
1  architecture rtl2 of bsk is
2    signal a: std_ulogic_vector(11 downto 0);
3  begin
4    process(clk)
5    begin
6      if rising_edge(clk) then
7        a <= din;
8      end if;
9    end process;
10   dout <= a or din;
11 end architecture rtl2;
```

Listing 4: Optimized synthesizable model

Let us denote `x[n]` the value of signal `x` at rising edge number `n` of `clk`, and `reg_x` the register with output `x`. One could think that there is a kind of redundancy between the three 12-bits registers but they correspond to different times: during the clock period between rising edges `n` and `n+1`, `reg_a` stores `din[n]`, `reg_b` stores `din[n-1]`, and `reg_dout` stores `din[n-2] OR din[n-3]`. Removing `reg_a`, for instance, by assigning

directly `b <= din`, would change the timing behaviour because during the clock period between rising edges `n` and `n+1`, the value of `dout` would now be `din[n-1] OR din[n-2]` instead of `din[n-2] OR din[n-3]`.

With the "optimized" version, the value of `dout` during the clock period between rising edges `n` and `n+1` is `din[n] OR din[n-1]`.

2. Problem: design and VHDL coding of a pipelined generic parity calculator

The goal of this problem is to design and code in synthesizable VHDL 2008 a generic, pipelined, parity calculator: `par`. It takes a multi-bits input `din` and computes its one-bit parity, that is, the exclusive OR of all bits of `din`. If `din` contains an even number of bits equal to `'1'`, its parity is `'0'`, else it is `'1'`. The bit width of `din` is greater or equal 2 and is a power of 2 (2, 4, 8,…)

`par` is generic; it has a generic parameter `n` of type `positive` which is the logarithm in base 2 of the bit width of the `din` input: if `n=1`, `din` is a two-bits vector, if `n=2`, `din` is a four-bits vector, …, if `n=10`, `din` is a 1024-bits vector, … Table 1 describes the interface of `par`.

Name	Direction	Bit width	Description
`clk`	input	1	clock
`srstn`	input	1	active low synchronous reset
`load`	input	1	load control signal
`din`	input	2^n	data input
`dso`	output	1	`'1'` when output valid
`do`	output	1	parity output

Table 1: Interface of `par`

As we need very high performance `par` is pipe-lined: the parity is computed in `n` successive clock cycles. When the `load` control signal is equal to `'1'` on a rising edge of the clock `clk`, `din` is stored in a 2^n-bits input register. During the following clock cycle the content of the input register is processed: each pair of bits is XOR-ed and the result is stored in a second 2^{n-1} bits register; during the following clock cycle each pair of bits of this first intermediate result is XOR-ed and the result is stored in another 2^{n-2} bits register; and so on until there is only one remaining result bit, which is stored in a last one-bit register. The `do` output is the output of this last one-bit register.

Thanks to this pipelined architecture a new input can be submitted at each clock cycle. But as there is not always a new input to process the `load` control signal indicates at which clock cycles new inputs are submitted.

The `dso` output is set to '`1`' by `par` during one clock cycle when a valid parity result is available at the `do` output. This way the environment can sample `do` on each rising edge of `clk` where `dso='1'`. Note: if `load='1'` at rising edge number x of `clk`, then `par` sets `dso` to `'1'` during the clock period between rising edges $x+n$ and $x+n+1$, such that `dso='1'` on rising edge $x+n+1$.

The active low **synchronous** reset `srstn` is used to reinitialize `par`: when `srstn='0'` on a rising edge of `clk`, all bits of the internal registers are set to `'0'`, and any ongoing computation is aborted.

2.1. Design of block and state diagrams

Draw a block diagram of `par` with `n=3` using the symbols represented on Figure 6. Clearly identify the registers, give their size in bits, name them and explain what their role is.

If you use state machines represent them as "other synchronous elements" in your block diagram, with named inputs and outputs. Draw their state diagrams, explain whether they are Moore or Mealy state machines, and provide a description of their role.

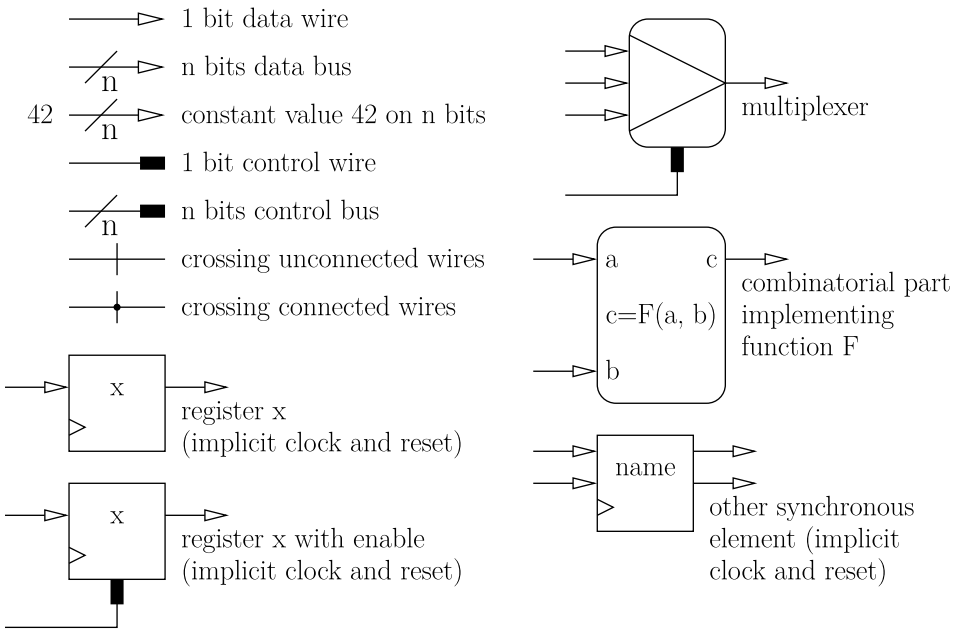

Figure 6: Symbols to design block diagrams

The block diagram of `par3` is shown on Figure 7 where the synchronous reset has been omitted for better readability. The `b = xh(a)` combinatorial block outputs the XOR of the two halves of its input bus `a`, which size must of course be even. The `dso` output is handled with the simple 4-stages r_d shift register which serial input is `load`. The other registers are the 4 data registers of the specification: r_3 is the 8-bits input register that stores `din`, r_2 is the 4-bits first intermediate register, r_1 is the 2-bits next intermediate register, and r_0 is the 1-bit last register, which output is `do`.

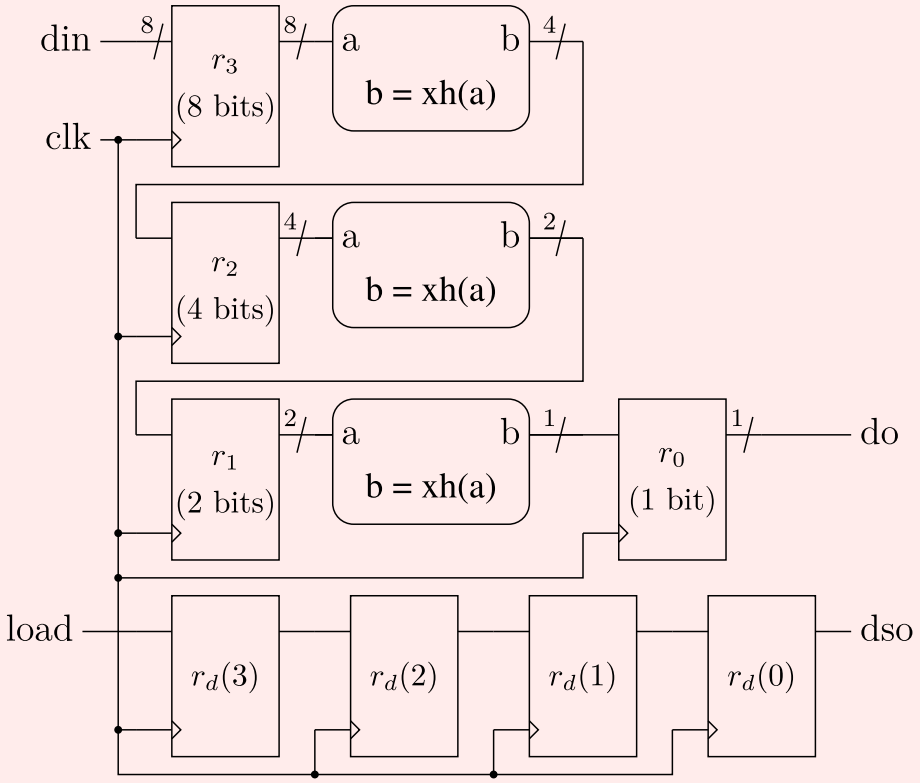

Figure 7: Block diagram of par3

2.2. VHDL coding for `n=3`

As an intermediate step, we consider `par3`, a non-generic version of `par` which `din` input is 8 bits wide. Code `par3` (entity and architecture) in synthesizable VHDL 2008. Do not use sub-designs and entity instantiations: code everything in the `par3` architecture. Add comments to explain non-obvious parts of your code.

The synthesizable VHDL 2008 model of `par3` is shown on Listing 5. The `b = xh(a)` combinatorial block of Figure 7 is implemented as the `xh` helper function.

Note how the function uses constants `size`, `lv` (a copy of the input parameter with known bounds), and `res`, to not depend on the actual bounds of the input parameter and to control the bounds of the returned vector.

The other parts are straightforward VHDL translation of the block diagram.

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

entity par3 is
  port(clk, srstn, load: in std_ulogic;
       din: in std_ulogic_vector(7 downto 0);
       do, dso: out std_ulogic);
end entity par3;

architecture rtl of par3 is

  signal rd: std_ulogic_vector(3 downto 0);
  signal r0: std_ulogic; -- could also be a 1-bit vector
  signal r1: std_ulogic_vector(1 downto 0);
  signal r2: std_ulogic_vector(3 downto 0);
  signal r3: std_ulogic_vector(7 downto 0);

  function xh(v: std_ulogic_vector) return std_ulogic_vector is
    constant size: natural := v'length;
    constant lv: std_ulogic_vector(size - 1 downto 0) := v;
    constant res: std_ulogic_vector(size / 2 - 1 downto 0) :=
      lv(size - 1 downto size / 2) xor lv(size / 2 - 1 downto 0);
  begin
    return res;
  end function xh;

begin

  process(clk)
  begin
    if rising_edge(clk) then
      if srstn = '0' then
        rd <= (others => '0');
        r0 <= '0'; -- scalar, not vector
        r1 <= (others => '0');
        r2 <= (others => '0');
        r3 <= (others => '0');
      else
        rd <= load & rd(3 downto 1);
        r0 <= xh(r1)(0); -- scalar, not vector
        r1 <= xh(r2);
        r2 <= xh(r3);
        r3 <= din;
      end if;
    end if;
  end process;

  do  <= r0; -- scalar, not vector
  dso <= rd(0);

end architecture rtl;
```

Listing 5: VHDL model of par3

2.3. VHDL coding for any value of `n`

Code `par` (entity and architecture) in synthesizable VHDL 2008, for **any** value of generic parameter `n`. Do not use sub-designs and entity instantiations: code everything in the `par` architecture. Add comments to explain non-obvious parts of your code.

The synthesizable VHDL 2008 model of `par` is shown on Listing 6. It is based on the same principles as `par3`, with data registers r_n, …, r_1, r_0, where r_n in the input register and r_0 is the output register. The main difference is that the number of data registers $(n+1)$ depends on the `n` generic parameter. They can thus not be declared individually.

Instead, we declare the array type `r_t`, with $n+1$ elements indexed from 0 to n where each element is a 2^n-bits vector, that is, the size of the input register, the largest of all. Then, we declare `r`, a signal of this type, to represents the outputs of all the data registers: `r(n)` is the output of r_n, …, and `r(0)` is the output of r_0.

Except for the input register, 2^n-bits is larger than needed but, as we use only the least significant bits, the synthesizer/optimizer will simply eliminate the useless register bits.

The data passing between the data registers is implemented as a for loop where the index `i` is that of the destination register in array `r`.

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

entity par is
  generic(n: positive);
  port(clk, srstn, load: in std_ulogic;
       din: in std_ulogic_vector(2**n - 1 downto 0);
       do, dso: out std_ulogic);
end entity par;

architecture rtl of par is

  function xh(v: std_ulogic_vector) return std_ulogic_vector is
    constant size: natural := v'length;
    constant lv: std_ulogic_vector(size - 1 downto 0) := v;
    constant res: std_ulogic_vector(size / 2 - 1 downto 0) :=
      lv(size - 1 downto size / 2) xor lv(size / 2 - 1 downto 0);
  begin
    return res;
  end function xh;

  type r_t is array(0 to n) of std_ulogic_vector(2**n - 1 downto 0);
  signal r: r_t;
  signal d: std_ulogic_vector(n downto 0);

begin

  process(clk)
  begin
    if rising_edge(clk) then
      if srstn = '0' then
        r <= (others => (others => '0'));
        d <= (others => '0');
      else
        for i in 0 to n - 1 loop
          r(i)(2**i - 1 downto 0) <= xh(r(i + 1)(2**(i + 1) - 1 downto 0));
        end loop;
        r(n) <= din;
        d    <= load & d(n downto 1);
      end if;
    end if;
  end process;

  do  <= r(0)(0); -- r(0) is a vector, not a scalar
  dso <= d(0);

end architecture rtl;
```

Listing 6: VHDL model of par

We can also declare only the necessary register bits, that is, $2^n + 2^{n-1} + ... + 2 + 1 = 2^{n+1} - 1$ bits:

```vhdl
signal r: std_ulogic_vector(2**(n+1) - 2 downto 0);
```

This is sufficient to implement all data registers as shown on Figure 8.

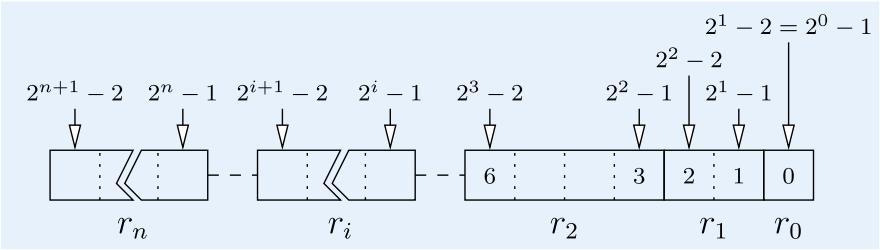

Figure 8: Registers mapping of the alternate model of par

From Figure 8 and some basic arithmetics we immediately see that r_i (with $0 \leq i \leq n$) is slice $2^{i+1}-2 \cdots 2^i-1$ of `r`. So, we map data register r_i to `r(2**(i+1) - 2 downto 2**i - 1)`. The input register r_n, for instance, is mapped to `r(2**(n+1) - 2 downto 2**n - 1)`, and the output register r_0 is mapped to `r(0)`. This does not rely on the synthesizer/optimizer to remove the unused register bits, but the for loop is a bit more complicated as Listing 7 shows.

```
architecture rtl2 of par is

  function xh(v: std_ulogic_vector) return std_ulogic_vector is
    constant size: natural := v'length;
    constant lv: std_ulogic_vector(size - 1 downto 0) := v;
    constant res: std_ulogic_vector(size / 2 - 1 downto 0) :=
      lv(size - 1 downto size / 2) xor lv(size / 2 - 1 downto 0);
  begin
    return res;
  end function xh;

  signal r: std_ulogic_vector(2**(n + 1) - 2 downto 0);
  signal d: std_ulogic_vector(n downto 0);

begin

  process(clk)
  begin
    if rising_edge(clk) then
      if srstn = '0' then
        r <= (others => '0');
        d <= (others => '0');
      else
        for i in 0 to n - 1 loop
          r(2**(i + 1) - 2 downto 2**i - 1) <=
            xh(r(2**(i + 2) - 2 downto 2**(i + 1) - 1));
        end loop;
        r(2**(n + 1) - 2 downto 2**n - 1) <= din;
        d <= load & d(n downto 1);
      end if;
    end if;
  end process;

  do  <= r(0);
  dso <= d(0);

end architecture rtl2;
```

Listing 7: Alternate VHDL model of par

2.4. Power optimization

Assume that, after a number of parity computations, there are no more inputs to submit to `par` for some (long) time. Will your design still consume power, even if there is nothing to compute? If no explain why. Else suggest ways to reduce this useless power consumption; do not code in VHDL, just explain what should be done to save power.

The design will still consume power if the `din` input changes, even if `load` remains zero. Indeed, the parity will still be computed. It is only `dso` that will remain zero to indicate that these parity outputs shall be ignored. But the toggling of all the internal wires will consume power. To prevent this we could sample `din` in the input register only if `load ='1'`. The power would then decrease because once all internal registers converged to their final value there would be no more wire toggling.

However, if the clock is still active, there will still be its own power consumption, unless a power saving mode is implemented and the clock generator stops the clock. But this last technique does not depend on our design.

VHDL QUICK REFERENCE CARD

Revision 2.2

()	Grouping	[ ]	Optional
{}	Repeated	|	Alternative
bold	As is	CAPS	User Identifier
italic	VHDL-1993		

1. LIBRARY UNITS

[{use_clause}]
entity ID **is**
  [**generic (**{ID : TYPEID [**:=** expr];}**);**]
  [**port (**{ID **: in | out | inout** TYPEID [**:=** expr];}**);**]
  [{declaration}]
[**begin**
  {parallel_statement}]
end [***entity***] ENTITYID;

[{use_clause}]
architecture ID **of** ENTITYID **is**
  [{declaration}]
begin
  [{parallel_statement}]
end [***architecture***] ARCHID;

[{use_clause}]
package ID **is**
  [{declaration}]
end [***package***] PACKID;

[{use_clause}]
package body ID **is**
  [{declaration}]
end [***package body***] PACKID;

[{use_clause}]
configuration ID **of** ENTITYID **is**
for ARCHID
  [{block_config | comp_config}]
end for;
end [***configuration***] CONFID;

use_clause::=
  library ID;
  [{**use** LIBID.PKGID[**. all** | DECLID];}]

block_config::=
  for LABELID
    [{block_config | comp_config}]
  end for;

comp_config::=
  for all | LABELID : COMPID
    (**use entity** [LIBID.]ENTITYID [**(** ARCHID **)**]
      [[**generic map (** {GENID **=>** expr ,} **)**]
       port map ({PORTID **=>** SIGID | *expr* ,}**)**];
    [**for** ARCHID
      [{block_config | comp_config}]
    end for;]
    end for;) |
    (**use configuration** [LIBID.]CONFID
      [[**generic map (**{GENID **=>** expr ,}**)**]
      port map ({PORTID **=>** SIGID | *expr*,}**)**];)
  end for;

2. DECLARATIONS

2.1. TYPE DECLARATIONS

type ID **is (** {ID,} **);**

type ID **is range** number **downto** | **to** number;

type ID **is array (** {range | TYPEID ,}**) of** TYPEID;

type ID **is record**
  {ID : TYPEID;}
end record;

type ID **is access** TYPEID;

type ID **is file of** TYPEID;

subtype ID **is** SCALARTYPID **range** range;

subtype ID **is** ARRAYTYPID**(** {range,}**);**

subtype ID **is** RESOLVFCTID TYPEID;

range ::=
  (integer | ENUMID **to** | **downto** integer | ENUMID) |
  (OBJID'[**reverse_**]**range**) | (TYPEID **range <>**)

2.2. OTHER DECLARATIONS

constant ID : TYPEID **:=** expr;

[***shared***] **variable** ID : TYPEID [**:=** expr];

signal ID : TYPEID [**:=** expr];

file ID : TYPEID (**is in** | **out** string;) |
  (***open read_mode*** | ***write_mode*** |
   append_mode is *string;*)

alias ID : TYPEID **is** OBJID;

attribute ID : TYPEID;

attribute ATTRID **of** OBJID | **others** | **all** : class **is** expr;

class ::=
  entity | **architecture** | **configuration** |
  procedure | **function** | **package** | **type** |
  subtype | **constant** | **signal** | **variable** |
  component | **label**

component ID [***is***]
  [**generic (** {ID : TYPEID [**:=** expr];} **);**]
  [**port (**{ID **: in** | **out** | **inout** TYPEID [**:=** expr];}**);**]
end component [*COMPID*];

[***impure*** | ***pure***] **function** ID
  [**(** {[**constant** | **variable** | **signal** | ***file***] ID :
    [**in**]TYPEID [**:=** expr];}**)**]
  return TYPEID [**is**
begin
  {sequential_statement}
end [***function***] ID];

procedure ID[({[**constant** | **variable** | **signal**] ID :
    in | **out** | **inout** TYPEID [**:=** expr];}**)**]
[**is begin**
  [{sequential_statement}]
end [***procedure***] ID];

for LABELID | **others** | **all** : COMPID **use**
  (**entity** [LIBID.]ENTITYID [**(** ARCHID **)**]) |
  (**configuration** [LIBID.]CONFID)
    [[**generic map (** {GENID **=>** expr,} **)**]
     port map ({PORTID **=>** SIGID | *expr*,} **)**];

3. EXPRESSIONS

expression ::=
  (relation **and** relation) | (relation **nand** relation) |
  (relation **or** relation) | (relation **nor** relation) |
  (relation **xor** relation) | (*relation* ***xnor*** *relation*)

relation ::= shexpr [relop shexpr]

shexpr ::= sexpr [*shop sexpr*]

sexpr ::= [**+|-**] term {addop term}

term ::= factor {mulop factor}

factor ::=
  (prim [****** prim]) | (**abs** prim) | (**not** prim)

prim ::=
  literal | OBJID | OBJID'ATTRID | OBJID({expr,})
  | OBJID(range) | ({[choice [{**|** choice}] **=>**] expr,})
  | FCTID({[PARID **=>**] expr,}) | TYPEID'(expr) |
  TYPEID(expr) | **new** TYPEID['(expr)] | **(** expr **)**

choice ::= sexpr | range | RECFID | **others**

3.1. OPERATORS, INCREASING PRECEDENCE

logop	**and** | **or** | **xor** | **nand** | **nor** | ***xnor***
relop	**=** | **/=** | **<** | **<=** | **>** | **>=**
shop	***sll*** | ***srl*** | ***sla*** | ***sra*** | ***rol*** | ***ror***
addop	**+** | **-** | **&**
mulop	***** | **/** | **mod** | **rem**
miscop	****** | **abs** | **not**

© 1995-2000 Qualis Design Corporation

4. SEQUENTIAL STATEMENTS

wait [**on** {SIGID,}] [**until** expr] [**for** time];

assert expr
  [**report** string]
  [**severity note | warning | error | failure**];

report *string*
  [severity note | warning | error | failure];

SIGID **<=** [**transport**] | [[***reject*** *TIME*] ***inertial***]
  {expr [**after** time],};

VARID **:=** expr;

PROCEDUREID[({[PARID **=>**] expr,})];

[*LABEL:*] **if** expr **then**
  {sequential_statement}
[{**elsif** expr **then**
  {sequential_statement}}]
[**else**
  {sequential_statement}]
end if [*LABEL*];

[*LABEL:*] **case** expr **is**
{**when** choice [{**|** choice}] **=>**
  {sequential_statement}}
end case [*LABEL*];

[LABEL:] [**while** expr] **loop**
  {sequential_statement}
end loop [LABEL];

[LABEL:] **for** ID **in** range **loop**
  {sequential_statement}
end loop [LABEL];

next [LOOPLBL] [**when** expr];

exit [LOOPLBL] [**when** expr];

return [expression];

null;

5. PARALLEL STATEMENTS

LABEL: **block** [***is***]
  [**generic (** {ID : TYPEID;} **);**
    [**generic map (** {[GENID **=>**] expr,} **);**]]
  [**port (** {ID : **in | out | inout** TYPEID } **);**
    [**port map (** {[PORTID **=>**] SIGID | *expr*,} **)**];]
  [{declaration}]
begin
  [{parallel_statement}]
end block [LABEL];

[LABEL:] [***postponed***] **process** [({SIGID,})]
  [{declaration}]
begin
  [{sequential_statement}]
end [***postponed***] **process** [LABEL];

[LBL:] [***postponed***] PROCID({[PARID **=>**] expr,});

[LABEL:] [***postponed***] **assert** expr
  [**report** string]
  [**severity note | warning | error | failure**];

[LABEL:] [***postponed***] SIGID **<=**
  [**transport**] | [[***reject*** *TIME*] ***inertial***]
  [{{expr [**after** TIME,]} | ***unaffected*** **when** expr **else**}]
  {expr [**after** TIME,]} | ***unaffected***;

[LABEL:] [***postponed***] **with** expr **select**
  SIGID **<=** [**transport**] | [[***reject*** *TIME*] ***inertial***]
    {{expr [**after** TIME,]} | ***unaffected***
      when choice [{**|** choice}]};

LABEL: COMPID
    [[**generic map (** {GENID **=>** expr,})]
     port map ({[PORTID **=>**] SIGID | *expr*,} **)**];

LABEL: ***entity*** *[LIBID.]ENTITYID [(ARCHID)]*
    *[[**generic map (** {GENID **=>** expr,})]*
     port map (*{[PORTID **=>**] SIGID | expr,})];*

LABEL: ***configuration*** *[LIBID.]CONFID*
    *[[**generic map (** {GENID **=>** expr,})]*
     port map (*{[PORTID **=>**] SIGID | expr,})];*

LABEL: **if** expr **generate**
  [{parallel_statement}]
end generate [LABEL];

LABEL: **for** ID **in** range **generate**
  [{parallel_statement}]
end generate [LABEL];

6. PREDEFINED ATTRIBUTES

TYPID'**base**	Base type
TYPID'**left**	Left bound value
TYPID'**right**	Right-bound value
TYPID'**high**	Upper-bound value
TYPID'**low**	Lower-bound value
TYPID'**pos**(expr)	Position within type
TYPID'**val**(expr)	Value at position
TYPID'**succ**(expr)	Next value in order
TYPID'**pred**(expr)	Previous value in order
TYPID'**leftof**(expr)	Value to the left in order
TYPID'**rightof**(expr)	Value to the right in order
*TYPID'**ascending***	*Ascending type predicate*
*TYPID'**image**(expr)*	*String image of value*
*TYPID'**value**(string)*	*Value of string image*
ARYID'**left**[(expr)]	Left-bound of [nth] index
ARYID'**right**[(expr)]	Right-bound of [nth] index
ARYID'**high**[(expr)]	Upper-bound of [nth] index
ARYID'**low**[(expr)]	Lower-bound of [nth] index
ARYID'**range**[(expr)]	'left down/to 'right
ARYID'**reverse_range**[(expr)]	'right down/to 'left
ARYID'**length**[(expr)]	Length of [nth] dimension
*ARYID'**ascending**[(expr)]*	*'right >= 'left ?*
SIGID'**delayed**[(TIME)]	Delayed copy of signal
SIGID'**stable**[(TIME)]	Signals event on signal
SIGID'**quiet**[(TIME)]	Signals activity on signal
SIGID'**transaction**	Toggles if signal active

SIGID'**event**	Event on signal ?
SIGID'**active**	Activity on signal ?
SIGID'**last_event**	Time since last event
SIGID'**last_active**	Time since last active
SIGID'**last_value**	Value before last event
*SIGID'**driving***	*Active driver predicate*
*SIGID'**driving_value***	*Value of driver*
*OBJID'**simple_name***	*Name of object*
*OBJID'**instance_name***	*Pathname of object*
*OBJID'**path_name***	*Pathname to object*

7. PREDEFINED TYPES

BOOLEAN	True or false
INTEGER	32 or 64 bits
NATURAL	Integers >= 0
POSITIVE	Integers > 0
REAL	Floating-point
BIT	'0', '1'
BIT_VECTOR(NATURAL)	Array of bits
CHARACTER	7-bit ASCII
STRING(POSITIVE)	Array of characters
TIME	hr, min, sec, ms, us, ns, ps, fs
DELAY_LENGTH	*Time >= 0*

8. PREDEFINED FUNCTIONS

NOW	Returns current simulation time
DEALLOCATE(ACCESSTYPOBJ)	Deallocate dynamic object
FILE_OPEN*([status], FILEID, string, mode)*	*Open file*
FILE_CLOSE*(FILEID)*	*Close file*

9. LEXICAL ELEMENTS

Identifier ::= letter { [underline] alphanumeric }

decimal literal ::=integer [**.** integer] [**E**[**+|-**] integer]

based literal ::=
  integer **#** hexint [**.** hexint] **#** [**E**[**+|-**] integer]

bit string literal ::= **B|O|X** " hexint "

comment ::= **--** comment text

Qualis Design Corporation

Elite Training / Consulting in Reuse and Methodology

Phone: +1-503-670-7200 FAX: +1-503-670-0809
E-mail: info@qualis.com com Web:www.qualis.com