RTL Digital Design

The subject of this tutorial is the design and prototyping of a finite impulsive response numerical filter (FIR filter). In particular, we will trace all the project steps necessary to arrive from high-level specifications (system level) to a prototype mapped on FPGA, whose behaviour will be observed by means of laboratory measurements. The activity includes:

- Specification analysis: understanding the features implemented by the filter (essentially low-pass filtering and decimation), understanding the communication protocols through which the filter interfaces with the external environment, understanding the proposed hardware architecture.
- RTL design of the filter by writing synthesizeable VHDL code.
- Preparing a testbench to verify that the implementation is correct by using functional simulation and validation.
- Synthesis and post-synthesis simulation. Static timing analysis.
- FPGA mapping (Xilinx Virtex-E) and P&R. Static timing analysis. Post-mapping/post-P&R simulation with back-annotation of gate delays.

An integral part of this tutorial is the acquisition of a good degree of knowledge in the use of digital design CAD tools widely used in industrial contexts. In particular, the entire path from RTL implementation to FPGA mapping will be articulated with reference to Xilinx's ISE 6.2 environment. Specific VHDL and synthesis applications – Modelsim 6.0a and Synplify Pro 7.3 respectively – will also be accessible directly from the ISE environment.

Each group will be asked to perform the following activities:

- The RTL-level hardware implementation of some filter functions, by completing the filter's synthesized VHDL code.
- Refining the VHDL testbench prepared for the functional simulation of the filter by adding specific test portions.
- Debugging the unit by compiling code, run functional simulation, and output analysis.
- Synthesis of the filter, emphasizing in particular the aspects related to the correct setting of the synthesizer parameters and the verification of the reports obtained with respect to different settings.
- Mapping on FPGA: you will have to prove that you understand and know how to manage the different steps that lead from the synthesized netlist to the corresponding binary programming file for the HW card.
- Detailed documentation of the work done (both as documentation of the unit and as documentation of the project choices made).

Specifications

In this tutorial, we want to design an 8-bit FIR filter, which has an finite impulse response at 10 coefficients and integrates a 1/10 decimation functionality of the output into the same unit. The filter must be programmable, in the sense that the coefficients a_i, with i=0,... ,9 must be able to be loaded from the outside. The operation of the filter must be stopped until the coefficients have all been loaded correctly. In addition, you can explicitly manage enabling/disabling the filter and updating the value of the coefficients themselves.
The filter I/O interface is assigned as follows:

SIGNAL	*DIRECTION*	*TYPE*	*DESCRIPTION*
Ck	In	std_logic	System Clock.
res_n	In	std_logic	Asynchronous reset. Low active.
Enable	In	std_logic	Unit enabling. Active high.
set_a	In	std_logic	Enable loading of coefficients. Active high.
a_in	In	std_logic	Serial port for loading coefficients.
x_in	In	std_logic_vector (7 down 0)	8-bit filter input.
y_out	Out	std_logic_vector (7 down 0)	8-bit output of the filter (with decimation 1/10).
ready	Out	std_logic	Signals to the control unit that the coefficient loading have completed successfully.
valid_out	Out	std_logic	Signals to the control unit that the filter output is valid.
loading	Out	std_logic	Signals to the control unit that the coefficients are being loaded serially.

The block implements combined filtering with finite impulse response and a decimation of 1/10. The latter functionality in the current implementation should be considered as an update of the output value that will take place only every 10 clock cycles. During the 10 cycles the unit will read 10 input values on which it will produce the new output value by weighing them appropriately with the 10 coefficients (this is a causal filter in which the output depends on the current value of the input and the 9 previous values; compared to an FIR filter without decimation in fact 9 "intermediate" values are overlooked between two successive outputs. In formulas, if the output of a 10 coefficient FIR filter can be expressed using the following linear finite difference equation with constant coefficients:

$$y_{FIR}[n] = \sum_{k=0}^{9} a_k x[n-k]$$

the decimated output will be equal to:

$$y\_out[n] = y_{FIR}[10n].$$

Note that the reference frequency for the production of the decimated output y_out will also be considered as 10 times lower than that of inputs reading; in this perspective, the previous one can be rewritten as follows:

$$y\_out[n] = y_{FIR}[(n/10)*10], \forall n,$$

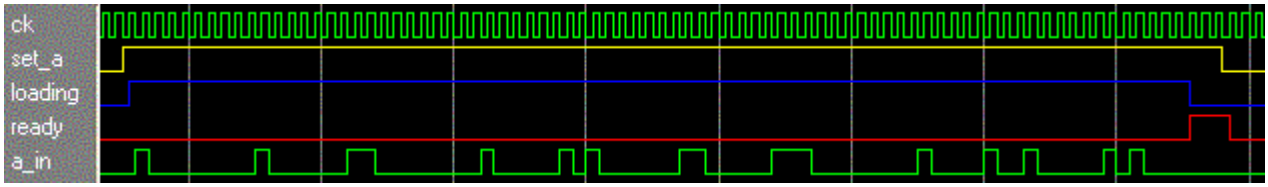
where the arithmetic operators designate operations *between integers* (especially integer division with truncation).

The unit handles 8-bit fixed-point data, meaning it reads 8-bit inputs and produces an 8-bit filtered output that is also decimated. However, for the implementation of arithmetic data-path, a 16-bit fixed-point arithmetic is used. Doubling the size of the data in the computational core[1] of the unit has the effect of reducing the total error on the output resulting from the accumulation of the truncation errors in the multiplication of operands to n bits (the result of multiplication of factors represented with n bits requires a 2n-bit representation for there to be no loss of information). In essence, the accumulation process (sum of products) that underlies any numerical filtering and contributes to the construction of the output value will work on the 16-bit data, while truncation will occur only and directly on the individual outputs decimated, once calculated. Finally, choosing fixed-point representation treats the data substantially identically to the entire representation (this is a positional notation where the decimal point is fixed upstream and once and for all) with the exception of the explicit handling of positive and negative overflow.

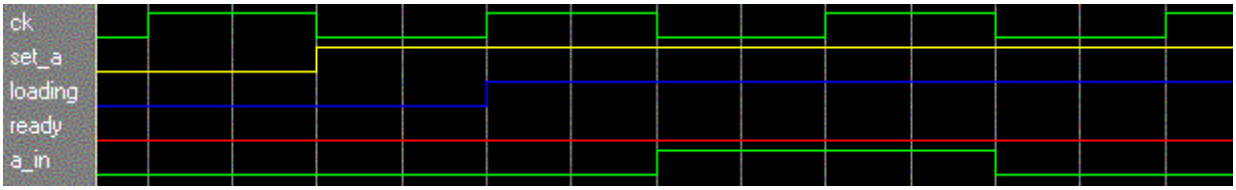
The implementation should be designed to handle sufficiently high processing speeds[2]. However, the realization of decimation has allowed to develop an ad hoc architecture that parallels multiple data-paths arithmetic, each of which can work at an actual clock frequency that is a submultiple of that system. This allows you to relax timing constraints at critical computational units (such as multipliers). For more details, see the description of the filtering and decimation process.

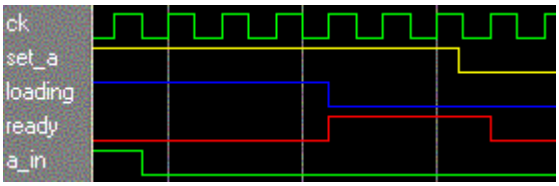
At the functional level, two distinct operating modes can be distinguished: the configuration of the unit on one side (loading of coefficients), the actual process of filtering the inputs and the production of the output decimated on the other. The specifications for each of these two modes of operation in both functional and architectural and protocol terms are as follows:

i) **Loading coefficients.** The filter works with a set of 10 coefficients that can be loaded from the outside using a serial input port. Through this port (a_in) an internal shift register of 8 x 10 = 80 bit is made accessible, which will be progressively filled until all 10 coefficient are fully loaded. The loading process is managed by a rather rigid communication protocol with the block's driving unit, and specified in terms of timing as follows [3] (the second and third figures representing a magnification of the behavior at the first and last clock cycles of the first):

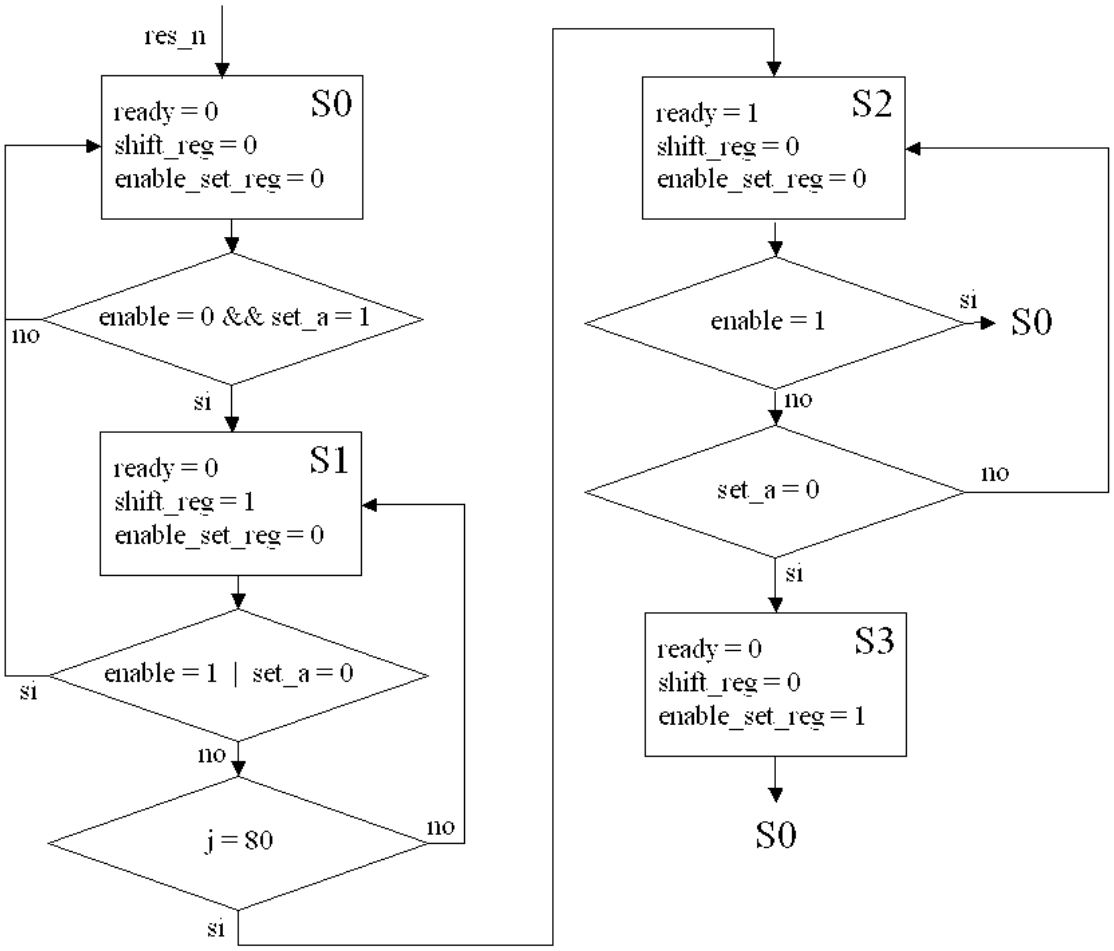


[1] It is also a well-established practice in dedicated industrial products such as signal digital processing (DSPs).
[2] During development, the specifications in terms of maximum frequency will be explored.
[3] Here and in the following, timing diagrams should be understood as functional specifications and as such without any reference to specific delay relationships between the different signals as they would result from their mapping on a particular technology.





The loading process is also controlled by a state machine whose ASM specification is as follows:

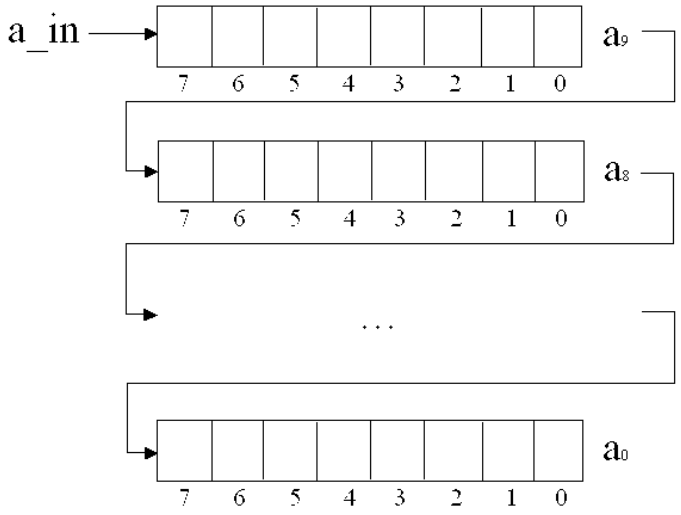


This is a synchronous Moore-type machine whose outputs have the following meaning:

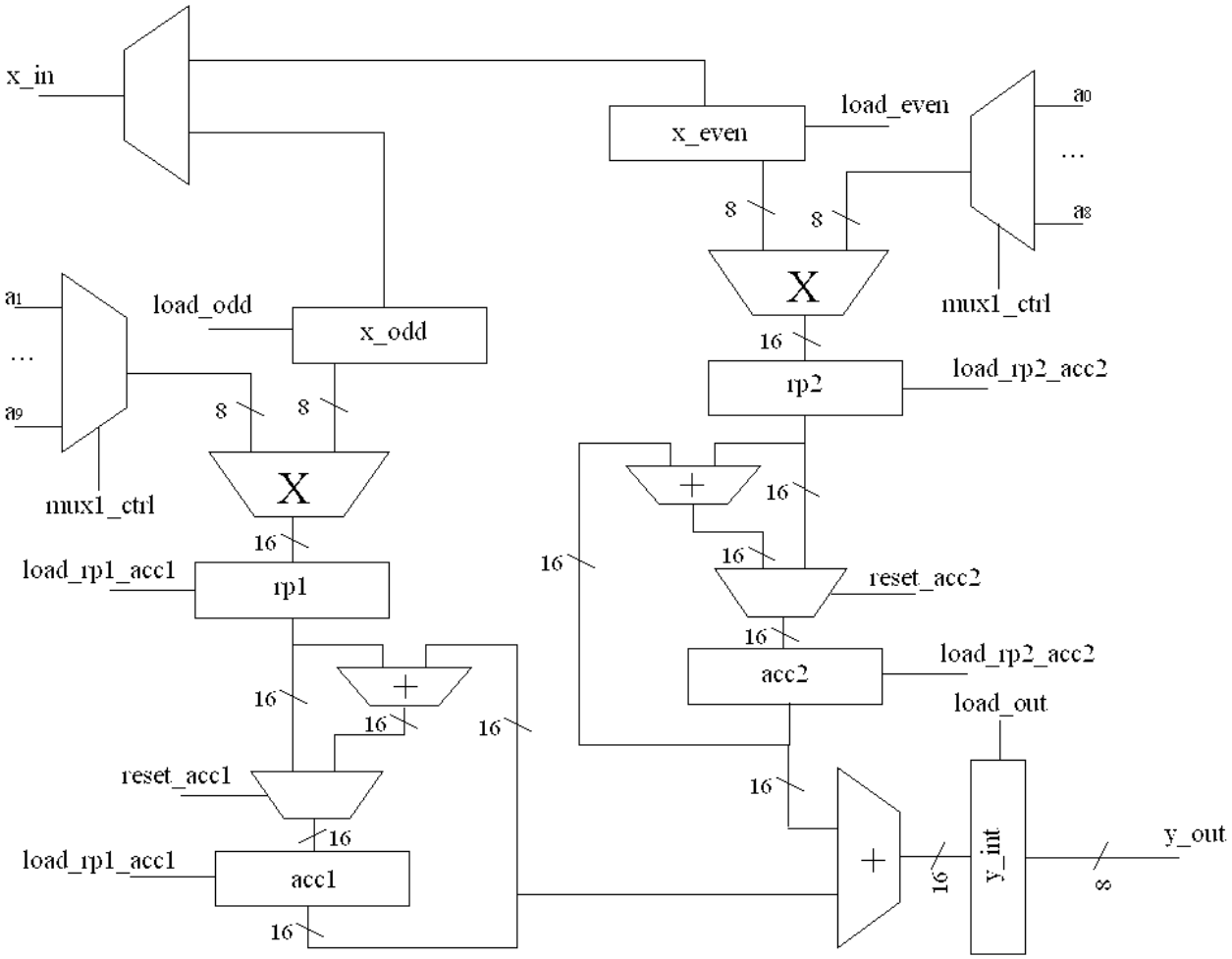
- *ready*: Asserted (active high) when the coefficient loading has been completed and released when the driver responds by deasserting the input set_a; corresponds to the output of the unit of the same name;
- *shift_reg*: Asserted (active high) acts as an enable for the shift register, which shifts its contents of a position at each clock cycle; functionally *coincides* with the "loading" output of the block.
- *enable_set_reg*: is an input of the filter data-path control machine that enables output production only when the coefficients have been loaded at least once fully and correctly (it is active high). The mechanism causes the filter not to start regardless of whether the system enable is enabled if the above condition is not verified (see Arithmetic data-path checking for more details).

The *variable j* models a counter that checks for the correct number of bits (80) corresponding to the full programming of the 10 coefficients (the count evolves parallel to the scrolling register fill state).

The structure (and internal connections) of the scrolling register is as follows:



ii) **Filtering and decimation**. The following figure shows the data-path architecture of the filter:

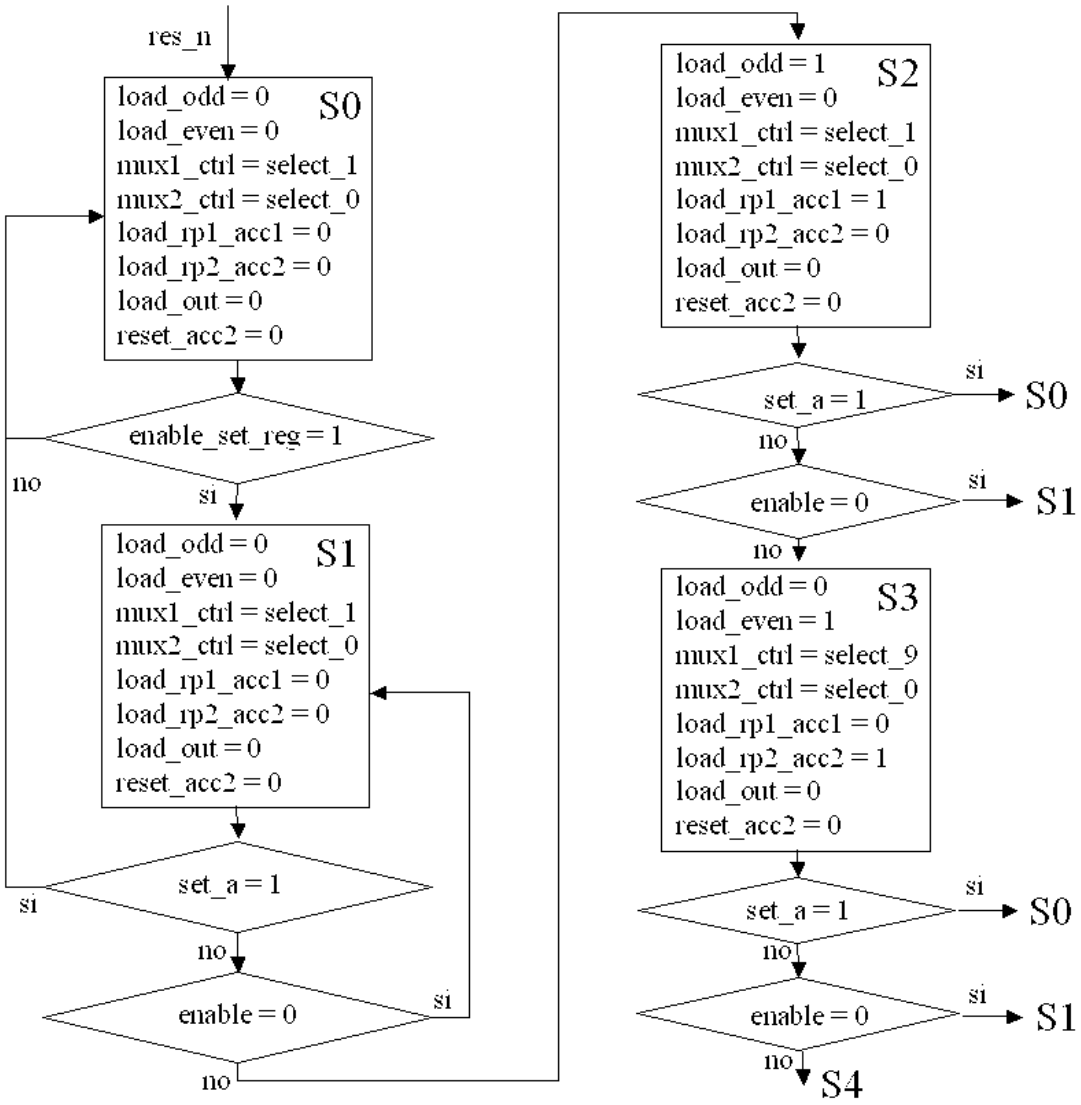


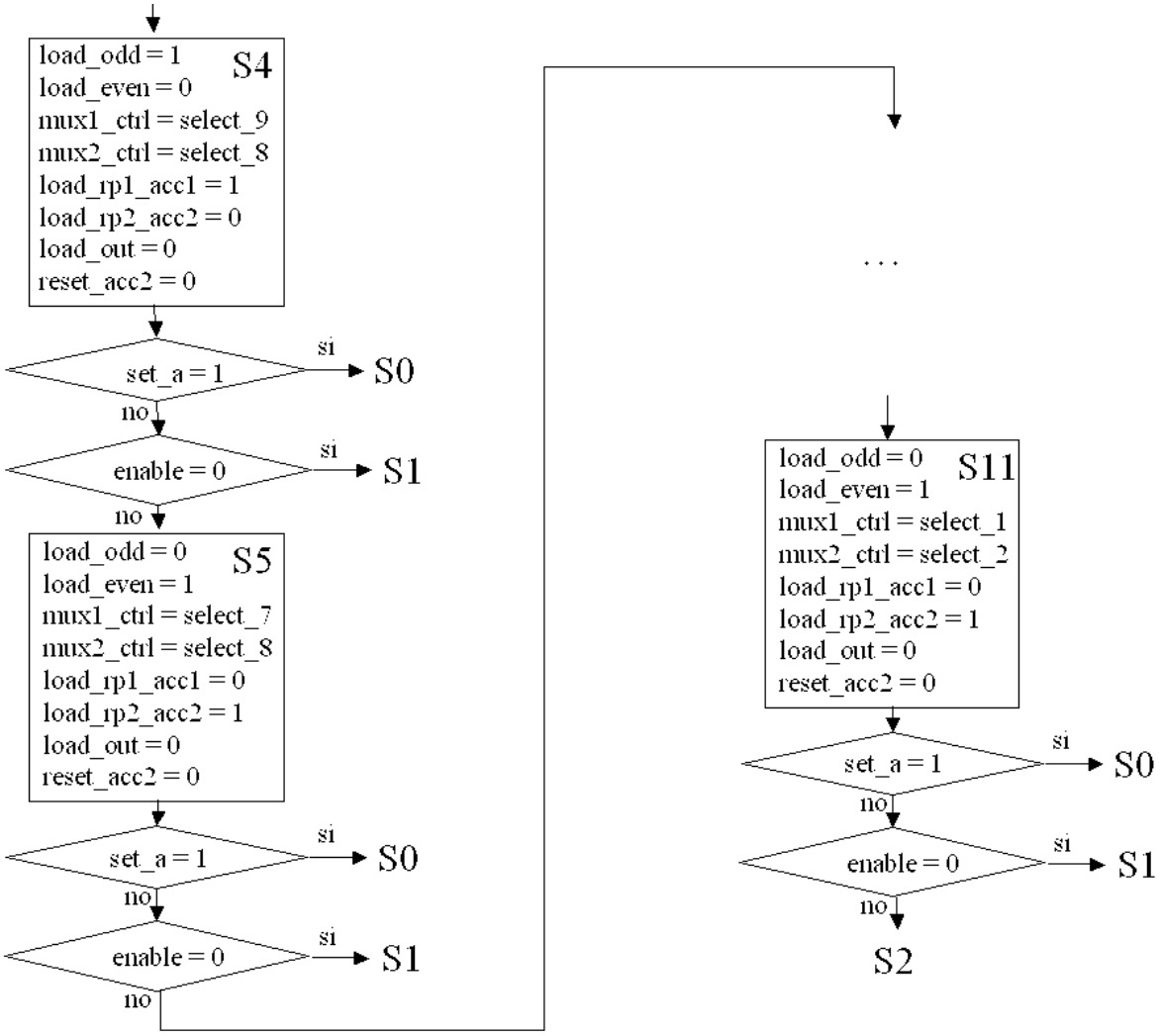
Two functionally identical branches that work in parallel but on opposite clock edges are distinguishable, which allows to assign an effective working frequency to each of them equal to ck/2. The key components are:

- 8-bit registers: x_odd, x_even,a_0, ...,a_9;
- 16-bit registers: rp1, rp2, acc1, acc2, y_int;
- 2 16-bit sdders;
- 2 8-bit multipliers (16-bit output);
- multiplexer;

Of these, all registers except y_int are loaded every 2 clock cycles, allowing, as mentioned, the intermediate logic to work at an actual frequency of half the clock frequency. Instead, the content of y_int is updated every 10 clock cycles and produces the decimated output (the truncation, as mentioned, is done after any arithmetic operation to minimize the error on the final result).

The figure also explicitly mentions the control signals of the different components (mux select and register enable signals). These signals are driven by a data-path control state machine (dedicated and distinct from the previous one that governs the block setup), the ASM specification of which is as follows:


res_n
load_odd = 0
load_even = 0
mux1_ctrl = select_1
mux2_ctrl = select_0
load_rp1_acc1 = 0
load_rp2_acc2 = 0
load_out = 0
reset_acc2 = 0
S0
enable_set_reg = 1
no
si
load_odd = 0
load_even = 0
mux1_ctrl = select_1
mux2_ctrl = select_0
load_rp1_acc1 = 0
load_rp2_acc2 = 0
load_out = 0
reset_acc2 = 0
S1
set_a = 1
si
no
enable = 0
si
no
load_odd = 1
load_even = 0
mux1_ctrl = select_1
mux2_ctrl = select_0
load_rp1_acc1 = 1
load_rp2_acc2 = 0
load_out = 0
reset_acc2 = 0
S2
set_a = 1
si
S0
no
enable = 0
si
S1
no
load_odd = 0
load_even = 1
mux1_ctrl = select_9
mux2_ctrl = select_0
load_rp1_acc1 = 0
load_rp2_acc2 = 1
load_out = 0
reset_acc2 = 0
S3
set_a = 1
si
S0
no
enable = 0
si
S1
no
S4



The S6, ..., S10 states are omitted in the diagram because the logic of the nextstate is identical to the explicitly represented cases while the output assignment logic is directly deductible on the model of the represented states. The output reset_acc2 is always 0 except at state 7, where it is 1; the output load_out is always 0 except at state 6, where it is 1. The reset_acc1 output is identical to the load_out signal.
The control state machine must be a synchronous Moore machine, ***with synchronized outputs;*** the latter feature is essential in order to actually take advantage of the working frequency halving at different points in the chain.

For example purposes, the following indicates what the computational core of the filter does at some rising edges of the system clock, assuming that the filtering operation has already started (enable asserted):

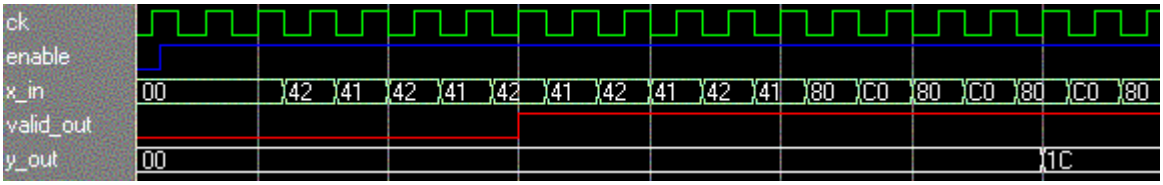
edge #1: Loads the register x_odd with input x_n;
loads the rp1 register (with the result of the product calculated in the *previous* **2** clock cycles););
loads the acc1 register (with the result of the sum calculated in the *previous* **2** cycles);
selects the a_n (n odd) weight, used in the *next* two cycles for the calculation of the $next_n$ product; ;
edge #2: Loads the x_even register with input x_{n+1};

loads the rp2 register (with the result of the product calculated in the *previous* **2** clock cycles););
loads the acc2 register (with the result of the sum calculated in the *previous* **2** cycles);
selects the weight a_{n+1} (n+1 even), used in the *next* two cycles for the calculation of the *next* product; ;

edge #3: loads the x_odd register with input x_{n+2};
loads the register rp1 (with the result of the product calculated in the *previous* **2** clock cycles , that is, $x_n * a_n$);
loads the register acc1 (with the result of the sum calculated in the *previous* **2** cycles , that is, accumulates the product $x_n * a_n$);
selects the weight a_{n+2} (n+2 odd), used in the *next* two cycles for the calculation of the *next* product; ;

...

The normal unit operation (filtering and decimation) is also controllable from the outside by a simple handshaking protocol, the timing of which is specified as follows:



The activation of the valid_out signal tells the unit that drives the filter that the value on the output y_out is valid, and is asserted (active high) after an appropriate number of clock cycles starting from the enabling (signal in the picture).

VHDL coding: The block must be represented at RTL level using simulated and **synthesizeable** VHDL code. You want a purely synchronous implementation, that is, without latches (the only sequential cells allowed are therefore flip-flops).

Testbench: A VHDL testbench is given. *The testbench must not obviously meet any synthesizer requirements*, but must ensure that the features of the block are as covered as possible. With this in mind, the suitability of the testbench will need to be validated and **documented** using an appropriate **code-coverage-verification** tool. The same testbench will then have to be applied, after back-annotating the delay information, at each level of the project from RTL coding to place&route.

VHDL ENCODING OVERVIEW
SYNTHESIZED ORIENTATA

Below are some "standard" ***synthesized*** VHDL structures, in the sense that they correspond to inference models of hardware structures recognized by any synthesizer:

- flip-flop (edge-triggered) with asynchronous reset:
  The classic inference pattern of a flip-flop of type D is using the following sequential process:

```
architecture <... >
...
signal Q, D, clock, reset :  std_logic;
...

begin

process (reset, clock)
begin
        if reset = ‘0’ then
                Q <= ‘0’;
        elsif clock'event 'event and clock ' '1' then
                Q <= D;
        end if;
end process;

end <... >
```

  Specifically, the code snippet shown corresponds to the implementation of a flip-flop D that is sensitive to the positive clock fronts and has active asynchronous reset low and output reset value of 0. Note that the sensitivity list of the sequential process, unlike what happens in the combinatorial processes – see In the following, the multiplexer inference model – contains *only* clock and reset signals.

- 8-bit register with asynchronous reset:
  The implementation of registers is basically similar the previous case, but the type of signals used is a std_logic_vector of appropriate size:

```
architecture <... >
...
signal clock, reset :  std_logic;
signal Q, D :  std_logic_vector (7 down 0);
...

begin

process (reset, clock)
begin
        if reset : '0' then
```

```
            Q <= (others => ‘0’);
        elsif clock'event 'event and clock ' '1' then
            Q <= D;
        end if;
end process;

end <... >
```

- Latch:
  Latch inference patterns are varied, but can be traced back to the simplest of them, the following:

```
architecture <... >
...
signal Q :  std_logic;
signal <other signals> :  std_logic;
...

begin

process (<all input signals to the process>)
begin
        if <condition on input signals> then
            Q <= <combinational operation o input signals>;
        end if;
end process;

end <... >
```

  The Q signal in this way will be mapped to the output of a latch, having control logic corresponding to the logical function mapping used in the assignment. Note that the process's sensitivity list contains *all* the signals used within the process as inputs.

- Multiplexer:
  A signal will be mapped to the output of a multiplexer whenever its value is defined in *all* possible branches of a conditional expression (if, case). Typical inference patterns are as follows, all equivalent to each other (all signals are of type std_logic, except sel which is of type std_logic_vector(1 downto 0)):

a)
```
process (sel, A, B, C, D)
Begin
    if sel : "00" then
            Y <= A;
    elsif sel : "01" then
            Y <= B;
    elsif sel : "10" then
            Y <= C;
    else, New10
            Y <= D;
    end if;
end process;
```

b) Y < A when sel : "00" else
B when sel : "01" else
C when sel : "10" else
D;

c) with sel select
Y < A when "00",
B when "01",
C when "10",
D when others;

d) process (sel, A, B, C, D)
Begin
houses sel is
when “00” => Y <= A;
when “01” => Y <= B;
when “10” => Y <= C;
when others => Y <= D;
end case;
end process;

It is essential that all branches of conditional expressions are specified (else/when others).

It is also noted that:

1) A typical error is **the unwanted** inference of asynchronous sequential logic when some branches of conditional expressions are not specified. In all models used for multiplexer inference, for example, the failure to specify else branches in if/when and when others in case/with ... select would have resulted in unwanted inference of one or more latches (regardless of the size of the driven signal).
It is important to note that this type of error is often NOT detected during functional simulation but gives rise to unexpected results directly at the synthesis level (however, the synthesizer itself usually indicates either as a warning or otherwise in the summary report the possible latch inference).
2) A typical error is the driving of the same signal at different points in the code (double drivers). This is reflected in the malfunction of the block, which can already be found at the functional simulation level but often with such effects that the cause is not immediately traceable. There are usually two cases:
    a. the signal in question is of a type for which a resolution function is *NOT* defined (see std_ulogic); the compiler itself reports the occurrence of the double driver (warning or error).
    b. The signal in question is of a type for which a resolution function is defined; the compiler *does NOT* report the presence of the double driver (different values possibly assigned to the signal in a concurrent manner would still be resolved according to the resolution matrix) and the anomaly is found directly in simulation.
3) Any initialization of signals at declaration level (before the architecture body) ***does NOT*** have any meaning from the synthesizer point of view (which will indicate with appropriate warning that it is ignoring initialization values). Using initialization values often simplifies the implementation of the functional testbench (basically it could allow you to "avoid" the modeling of a possible set-up phase of the block), but often results in unexpected malfunctions on the synthesized netlist (often, in fact, in the set-up phase).
4) Any delays set in the code using wait constructs have no physical meaning and are obviously ignored by the synthesis tool.

5) Direct assignment between two signals (i.e. a<=b) has the physical meaning of a hardware connection between the corresponding post-synthesis objects.