Sequential network examples in VHDL, hardware implementation of dataflow models

Tutorial 05 on Dedicated systems

Teacher: Giuseppe Scollo

University of Catania
Department of Mathematics and Computer Science
Graduate Course in Computer Science, 2018-19

Table of Contents

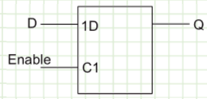
tutorial outline

this tutorial deals with:

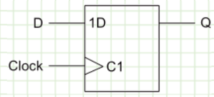
- sequential components:
  - latches, flip-flops, registers
  - counters, serial input registers
- hardware implementation of single-rate dataflow models
  example, Euclid's GCD algorithm:
  - SDF graph analysis
  - hardware implementation
- hardware pipelining:
  - throughput enhancement
  - warning about pipelining of SDF graphs with cycles
- lab experience
  - hardware implementation of a dataflow model

latches, flip-flops, registers

- *latch*: level-sensitive one-bit memory
- *flip-flop*: edge-triggered one-bit memory
- (parallel) register: bank of flip-flops



D latch symbol

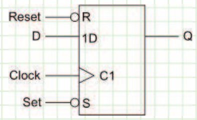


D type flip-flop

```
library ieee;
use ieee.std_logic_1164.all;
entity latch is
  port (
    d : in std_logic;
    en : in std_logic;
    q : out std_logic
  );
end entity latch;

architecture beh of latch is
begin
process (d, en) is
begin
  if (en = '1') then
    q <= d;
  end if;
end process;
end architecture beh;
```

```
library ieee;
use ieee.std_logic_1164.all;
entity dff is
  port (
    d : in std_logic;
    clk : in std_logic;
    q : out std_logic
  );
end entity dff;

architecture simple of dff is
begin
process (clk) is
begin
  if rising_edge(clk) then
    q <= d;
  end if;
end process;
end architecture simple;
```

```
library ieee;
use ieee.std_logic_1164.all;
entity register is
  generic ( n : natural := 8 );
  port (
    d : in std_logic_vector(n−1 downto 0);
    clk : in std_logic;
    nrst : in std_logic;
    load : in std_logic;
    q : out std_logic_vector(n−1 downto 0)
  );
end entity register;

architecture beh of register is
begin
process (clk, nrst) is
begin
  if (nrst = '0') then
    q <= (others => '0');
  elsif (rising_edge(clk) and (load = 1)) then
    q <= d;
  end if;
end process;
end architecture beh;
```

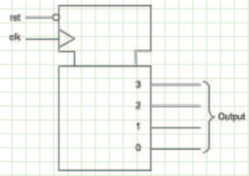


D-type flip-flop with asynchronous set and reset

counters, serial input registers

- counters: registers counting specified clock edges
  also used to implement timers
- serial input registers: partly similar to counters
  used for data input from serial lines, output is parallel

the use of *variables* easies the VHDL description in behavioural style



binary counter

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
entity counter is
  generic ( n : integer := 4 );
  port (
    clk : in std_logic;
    rst : in std_logic;
    output : out std_logic_vector(n−1 downto 0)
  );
end;
architecture simple of counter is
begin
  process(clk, rst)
    variable count : unsigned(n−1 downto 0);
  begin
    if rst = '0' then
      count := (others => '0');
    elsif rising_edge(clk) then
      count := count + 1;
    end if;
    output <= std_logic_vector(count);
  end process;
end;
```

```
library ieee;
use ieee.std_logic_1164.all;
entity shift_register is
  generic ( n : integer := 4 );
  port (
    clk : in std_logic;
    rst : in std_logic;
    din : in std_logic;
    q : out std_logic_vector(n−1 downto 0)
  );
end entity;
architecture simple of shift_register is
begin
  process(clk, rst)
    variable shift_reg : std_logic_vector(n−1 downto 0);
  begin
    if rst = '0' then
      shift_reg := (others => '0');
    elsif rising_edge(clk) then
      shift_reg := shift_reg(n−2 downto 0) & din;
    end if;
    q <= shift_reg;
  end process;
end architecture simple;
```


single-rate SDF graph to hardware

hardware implementation assumption:

single-rate SDF graphs, all actors operate at the same clock frequency

three implementation rules:

1. all actors are implemented as combinational circuits
2. all communication queues are implemented as wires (without storage)
3. each initial token on a communication queue is replaced by a register

two definitions:

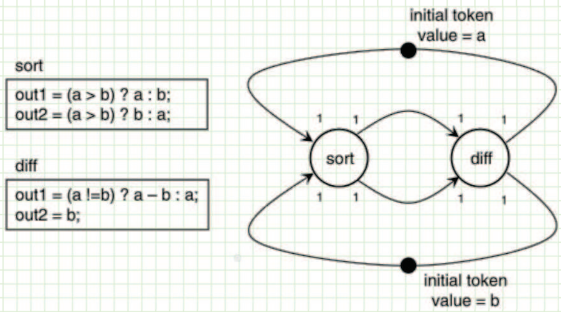
- *combinational path* in the SDF graph: cycle-free path with no initial tokens on it
- *critical path* in the SDF graph: combinational path that has maximum latency

maximum clock frequency for the circuit: reciprocal of latency through critical path

example: Euclid's GCD algorithm, SDF graph analysis

algorithm: at each step (a, b) is replaced by (|a-b|, min(a,b))

the pair converges to (GCD(a,b), GCD(a,b))



Schaumont, Figure 3.10 - Euclid's greatest common divisor as an SDF graph

PASS analysis:

$$G = \begin{bmatrix} +1 & -1 \\ +1 & -1 \\ -1 & +1 \\ -1 & +1 \end{bmatrix} \begin{matrix} \leftarrow edge(sort, diff) \\ \leftarrow edge(sort, diff) \\ \leftarrow edge(diff, sort) \\ \leftarrow edge(diff, sort) \end{matrix} \qquad \text{rank}(G) = 1 \qquad q_{PASS} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

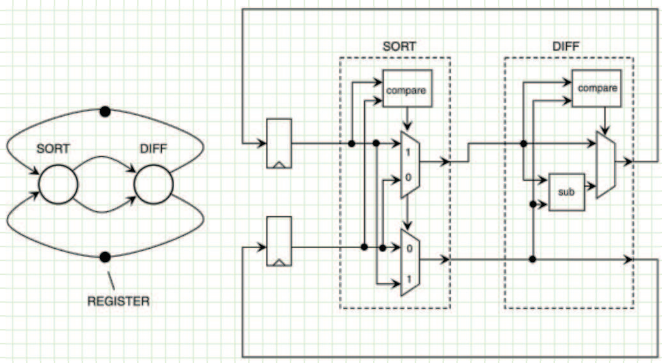
hardware implementation of Euclid's GCD algorithm

by the aforementioned three rules for a hardware implementation of the SDF model:

- two combinational circuits implement the actors
- a register is placed on each of the two connections from diff to sort

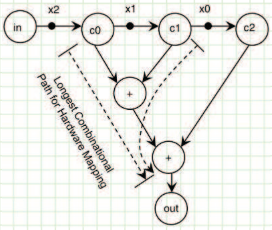
implementing the actors is a simple matter, by means of a few commonly used modules (multiplexers, comparators and a subtractor)

N.B. the HW diagram requires a bit of imagination and a correction

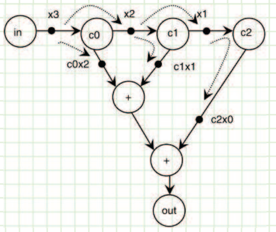


Schaumont, Figure 3.11 - Hardware implementation of Euclid's algorithm example of throughput enhancement by pipelining:

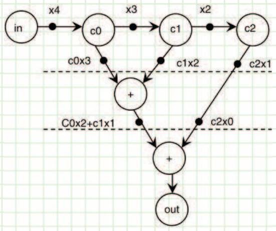
digital filter to produce a weighted sum: x0·c2+x1·c1+x2·c0



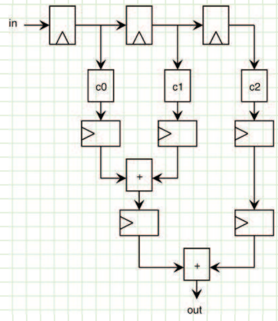
Schaumont, Figure 3.12
SDF graph of a simple moving-average application



Schaumont, Figure 3.13
Pipelining the moving-average filter by inserting additional tokens (1)



Schaumont, Figure 3.14
Pipelining the moving-average filter by inserting additional tokens (2)



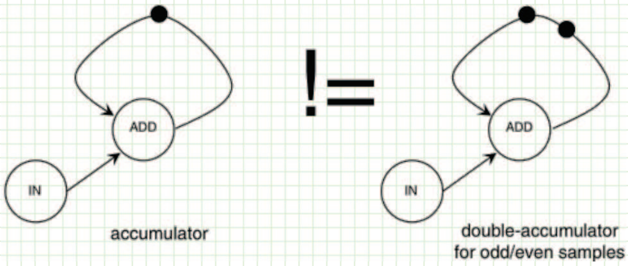
Schaumont, Figure 3.15
Hardware implementation of the moving-average filter

remarks:

- initial tokens here play the role of *delay* buffers in the definition of the pipelining transformation
- the SDF graph is cycle-free... (see next)

by introducing new tokens, pipelining may change the behaviour of an SDF graph

in particular, this may happen if additional tokens are introduced inside a loop, as this example shows:



Schaumont, Figure 3.16 - Loops in SDF graphs cannot be pipelined

in order to apply pipelining without changing the functional behaviour of an SDF graph with cycles, the additional tokens should be placed outside of any loop in the graph

for instance, on the input or output lines the circuit depicted in figure 3.11 implements the computational core of Euclid's GCD algorithm, yet it does not contain elements apt to signal the start and the end of the computation nor to distinguish inputs and output; the aims of this experience are: to extend that circuit to this purpose, to produce a VHDL description of it, to simulate it, and to implement it on the DE1-SoC FPGA

1. extend the schematic of the circuit in figure 3.11 with three input signals and two output signals:
   - a, b: the input data, 5-bit wide each
   - start: 1-bit input, to signal availability of the input data
   - gcd: the 5-bit output result
   - done: 1-bit output, to signal the end of computation and availability of the output result

   and with additional elements (flip-flop, multiplexers, maybe a comparator) useful to the stated purpose
2. produce a VHDL description of the designed circuit, either directly or through a Gezel description to be translated to VHDL by means of the fdlvhd program
3. create a Quartus project Euclid, assign it the produced .vhd files, compile, and simulate the behaviour of the circuit with a few input data pairs
4. create a new Quartus project Euclid_on_DE1SoC, assign it the previous .vhd files together with the 7-segment display decoder employed in lab tutorial 4 and a new top-level VHDL entity, composing the former one with an instance of the aforementioned decoder, while mapping the I/O signals to FPGA pins as follows:
   - a, b: SW9-5, SW4-0
   - start: not KEY1
   - RST: not KEY0
   - CLK: CLOCK_50 (50 MHz system clock)
   - done: LEDR0
   - gcd: LEDR1, HEX0
5. import the DE1-SoC pin assignments, compile, program the FPGA with the resulting .sof file, and test the functioning of the implementation with a few input data pairs

recommended readings:

Zwoliński, Ch. 6, Sect. 6.1-6.5.1

Schaumont, Ch. 3, Sect 3.2

readings for further consultation:

Schaumont, Ch. 3, Sect. 3.3

useful materials for the proposed lab experience
(source: Intel Corp. - FPGA University Program, 2016)

Debugging of VHDL Hardware Designs on Intel's DE-Series Boards - For Quartus Prime 16.1, Sect. 4.1, 6-8

VHDL sources:

examples in this presentation

examples in Zwoliński's book