ECE381(CAD), Lecture 14:

Design Example- pipeline

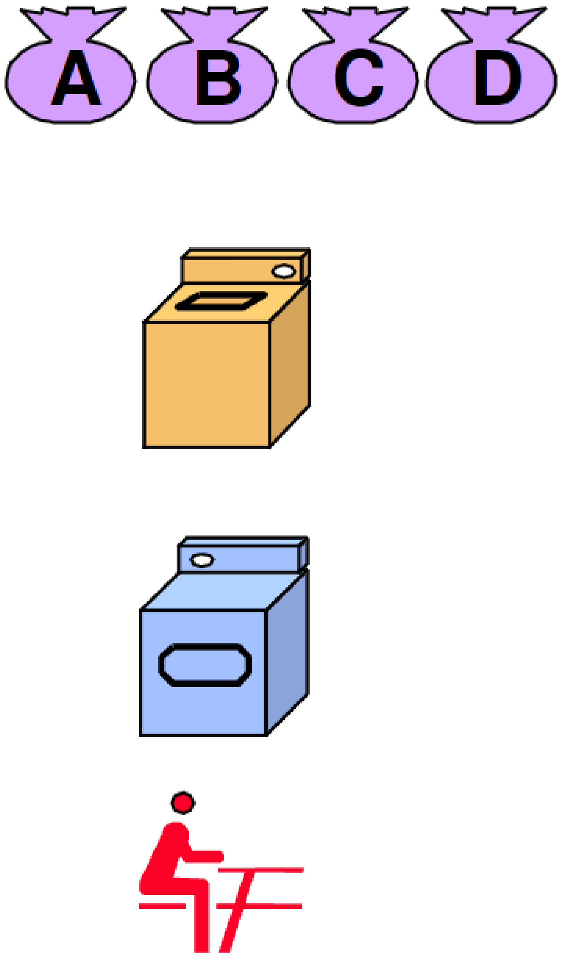
Mehdi Modarressi
Department of Electrical and Computer Engineering,
University of Tehran

Introduction

- Pipelining a datapath
- Reading:
  - “Designers Guide To VHDL”: chapter 6

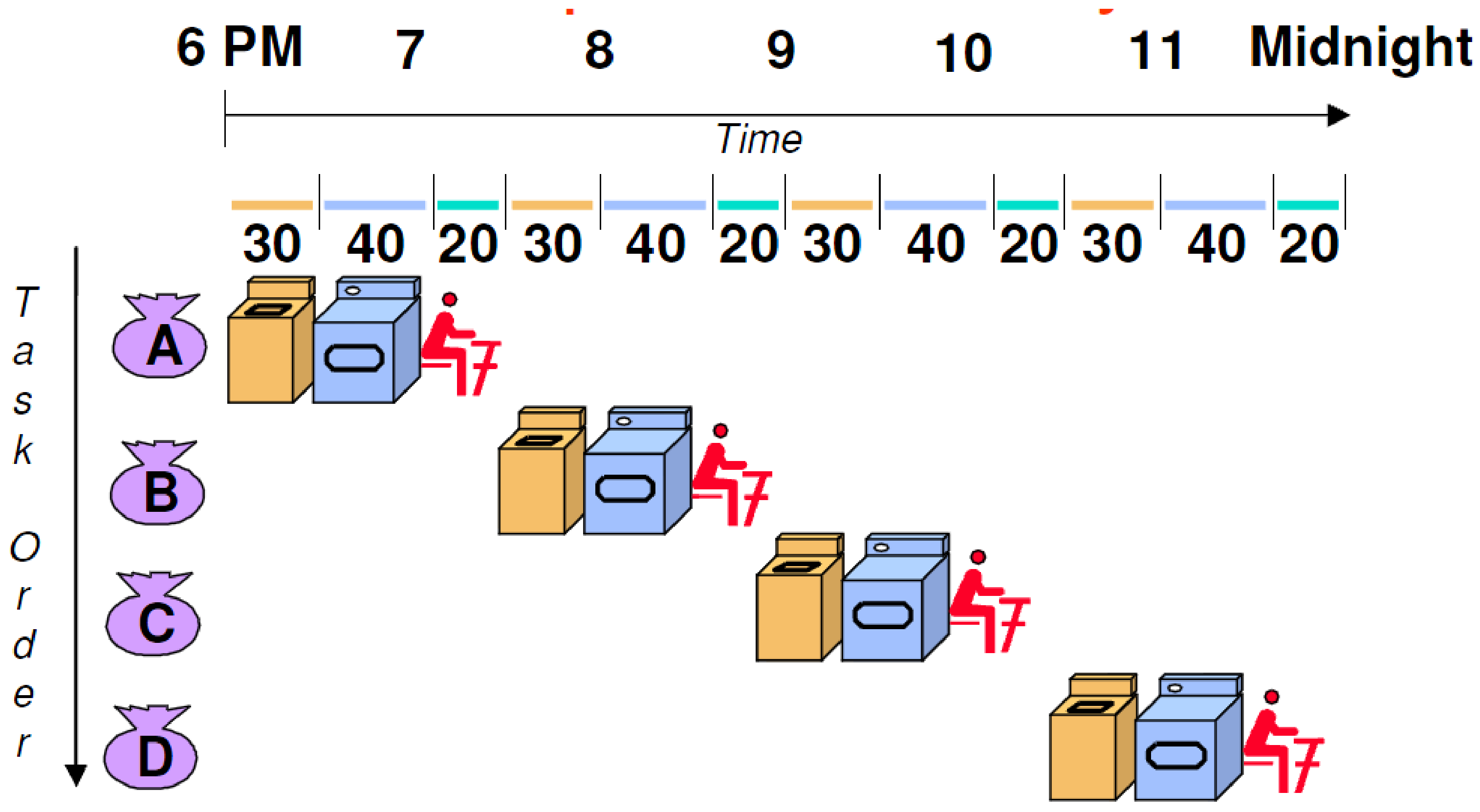
Pipeline

- Pipelining is Natural!
- A laundry with 4 laods
    - Washer takes 30 minutes
    - Dryer takes 40 minutes
    - Folder takes 20 minutes



Picture Source: ECE U530 by M. Leeser

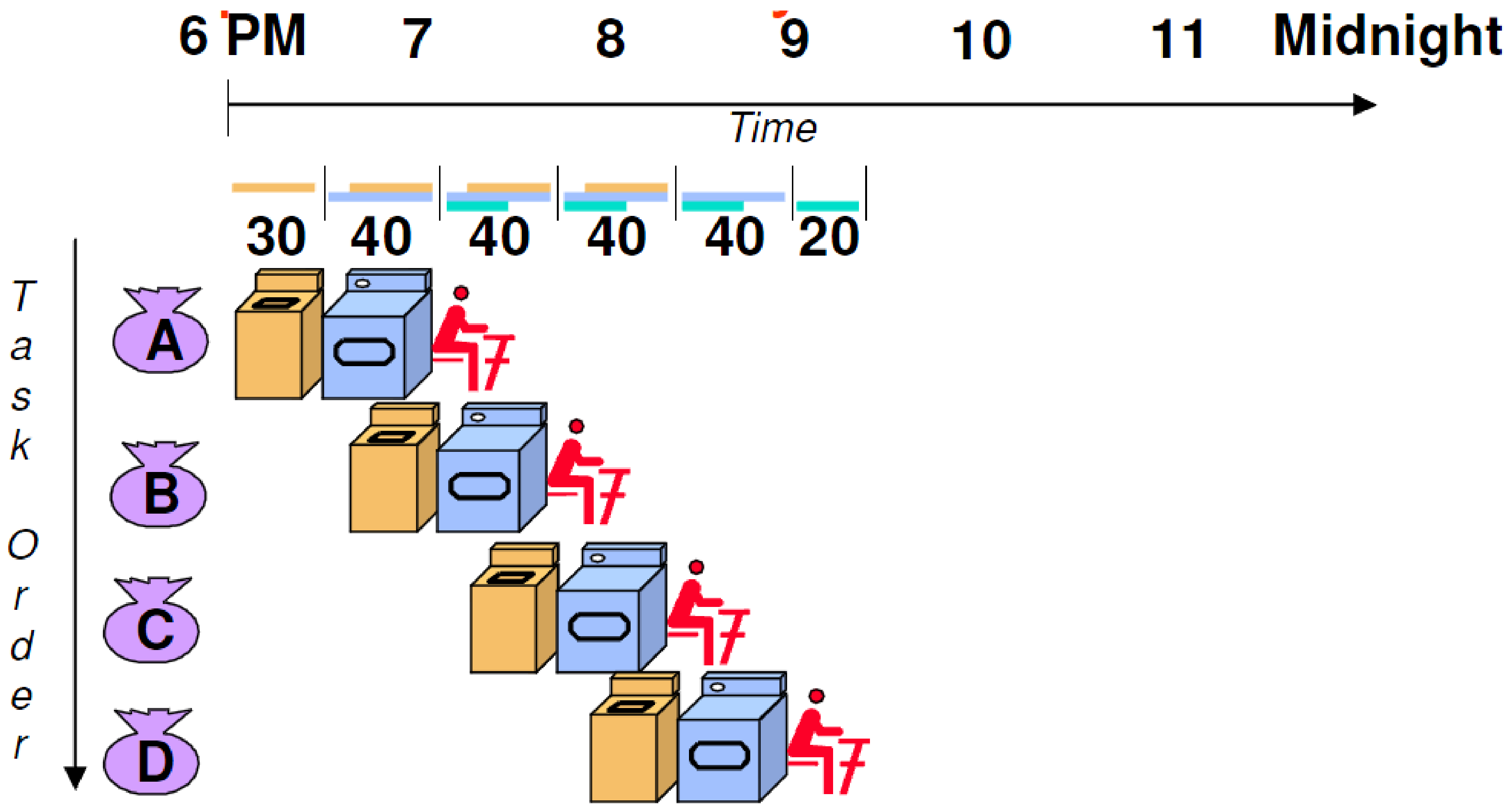
Sequential laundry



- Will take 6 hours

Source: ECE U530 by M. Leeser

Pipelined laundry



- Will take 3.5 hours

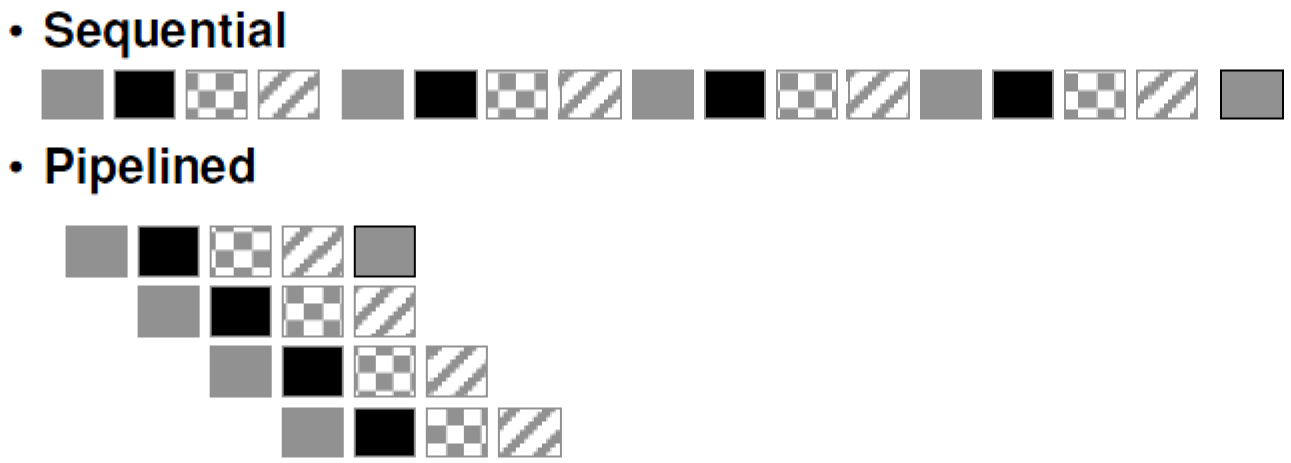
Source: ECE U530 by M. Leeser

Pipelining

- Pipelining doesn't reduce single task latency
  - It helps throughput of entire System
- Latency
  - Time from start of task to end of that task
- Throughput
  - The number of completed tasks per time unit
- Pipeline rate limited by slowest pipeline stage
- Potential speedup = Number pipe stages

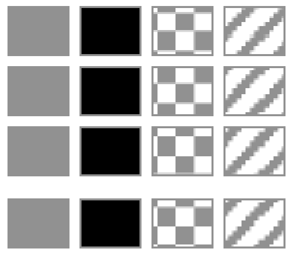
Pipeline

- Multiple tasks operating simultaneously using different resources



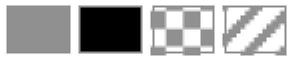
Pipeline vs parallel processing

- Parallel

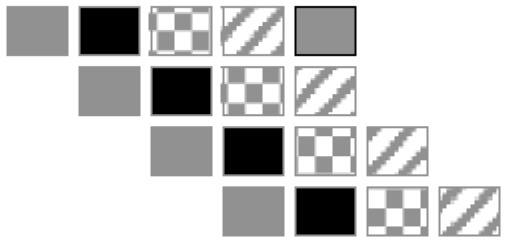


Parallelism requires more hardware

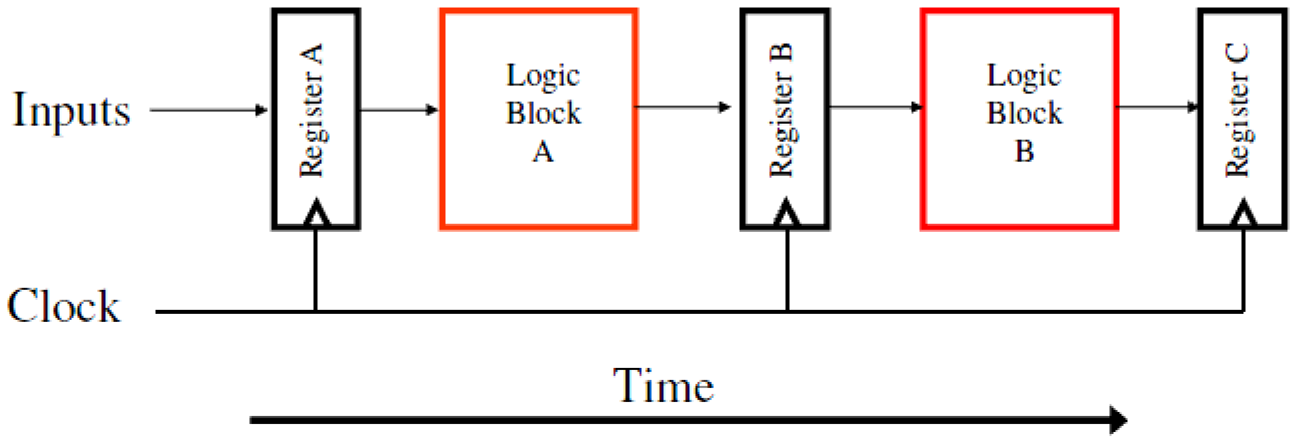
- Sequential and Pipelined have same hardware



- Pipelined schedule, not hardware units



Pipeline



- Datapath + staging registers

Pipeline example: MAC

- A pipelined multiplier accumulator (MAC) for a stream of complex numbers
  - Many digital signal processing algorithms
- A complex MAC operates on two sequences of complex numbers
  - Multiplies corresponding elements of the sequences and accumulates the sum of the products

$$\sum_{i=1}^{N} x_i y_i$$

Multiplication of x and y:

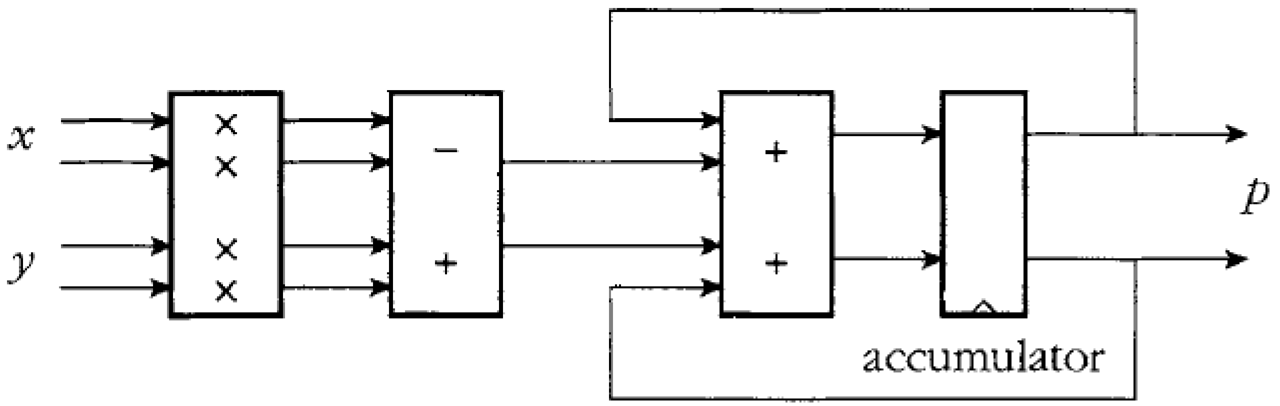
$$p_{\text{real}} = x_{\text{real}} \times y_{\text{real}} - x_{\text{imag}} \times y_{\text{imag}}$$
$$p_{\text{imag}} = x_{\text{real}} \times y_{\text{imag}} + x_{\text{imag}} \times y_{\text{real}}$$

sum of x and y:

$$s_{\text{real}} = x_{\text{real}} + y_{\text{real}}$$
$$s_{\text{imag}} = x_{\text{imag}} + y_{\text{imag}}$$

MAC diagram

- MAC without pipelining

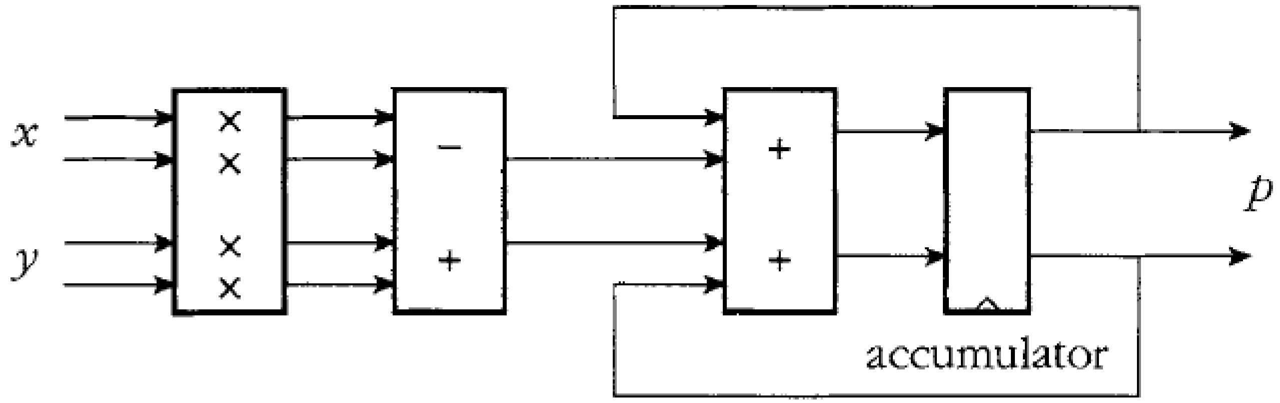


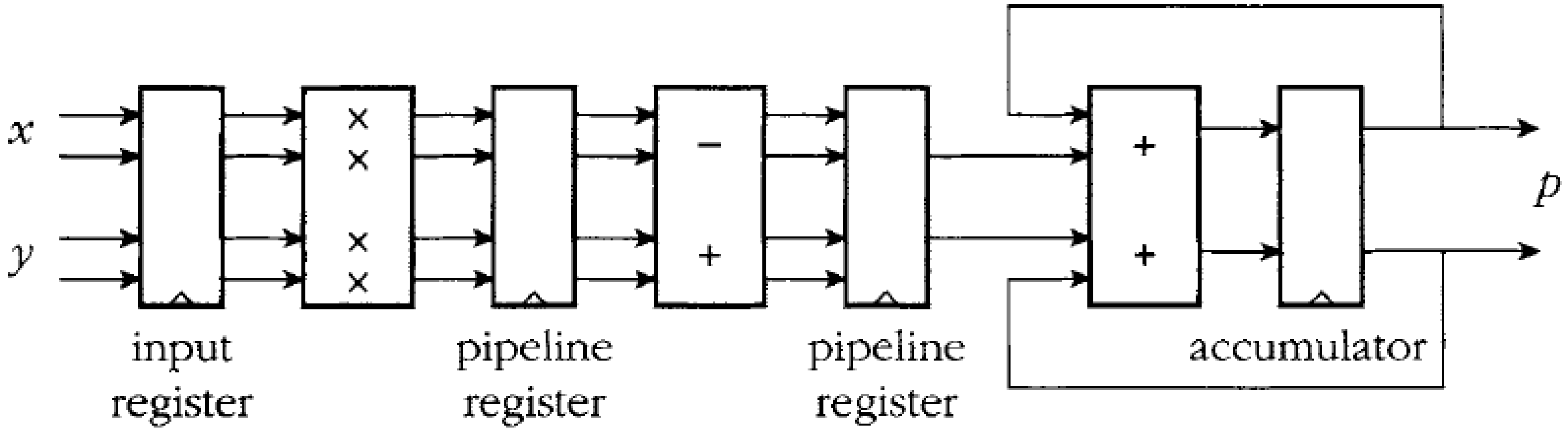
$$p_{\text{real}} = x_{\text{real}} \times y_{\text{real}} - x_{\text{imag}} \times y_{\text{imag}} \qquad s_{\text{real}} = x_{\text{real}} + y_{\text{real}}$$

$$p_{\text{imag}} = x_{\text{real}} \times y_{\text{imag}} + x_{\text{imag}} \times y_{\text{real}} \qquad s_{\text{imag}} = x_{\text{imag}} + y_{\text{imag}}$$

MAC diagram

- MAC with (bottom) and without (up) pipelining

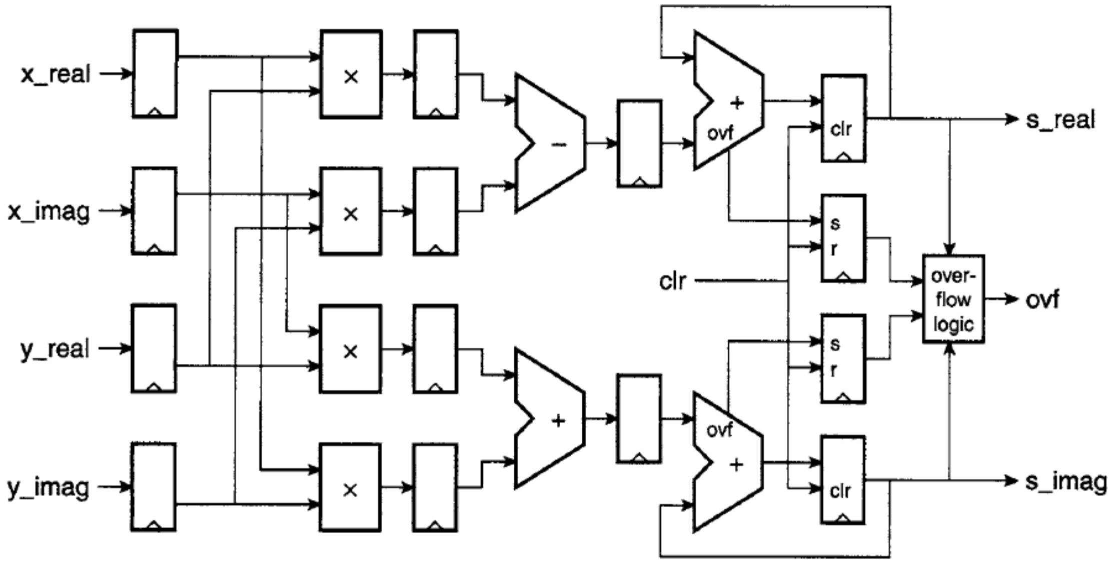




MAC entity declaration

```
library ieee;  use ieee.std_logic_1164.all;
entity mac is
      port ( clk, clr : in std_ulogic;
             x_real : in std_ulogic_vector(15 downto 0);
             x_imag : in std_ulogic_vector(15 downto 0);
             y_real : in std_ulogic_vector(15 downto 0);
             y_imag : in std_ulogic_vector(15 downto 0);
             s_real : out std_ulogic_vector(15 downto 0);
             s_imag : out std_ulogic_vector(15 downto 0);
             ovf : out std_ulogic );
end entity mac;
```

$$p_{real} = x_{real} \times y_{real} - x_{imag} \times y_{imag}$$
$$p_{imag} = x_{real} \times y_{imag} + x_{imag} \times y_{real}$$



More detailed structural diagram of MAC

Conversion entities

- The ports are *std_logic_vector*, but the operations are to be made on *real* numbers

```
library ieee;  use ieee.std_logic_1164.all;
entity mac is
      port ( clk, clr : in std_ulogic;
             x_real : in std_ulogic_vector(15 downto 0);
             x_imag : in std_ulogic_vector(15 downto 0);
             y_real : in std_ulogic_vector(15 downto 0);
             y_imag : in std_ulogic_vector(15 downto 0);
             s_real : out std_ulogic_vector(15 downto 0);
             s_imag : out std_ulogic_vector(15 downto 0);
             ovf : out std_ulogic );
end entity mac
```

```
library ieee;  use ieee.std_logic_1164.all;
entity to_fp is
      port ( vec : in std_ulogic_vector(15 downto 0);
             r : out real );
end entity to_fp;
```

```
library ieee;  use ieee.std_logic_1164.all;
entity to_vector is
      port ( r : in real;
             vec : out std_ulogic_vector(15 downto 0) );
end entity to_vector;
```

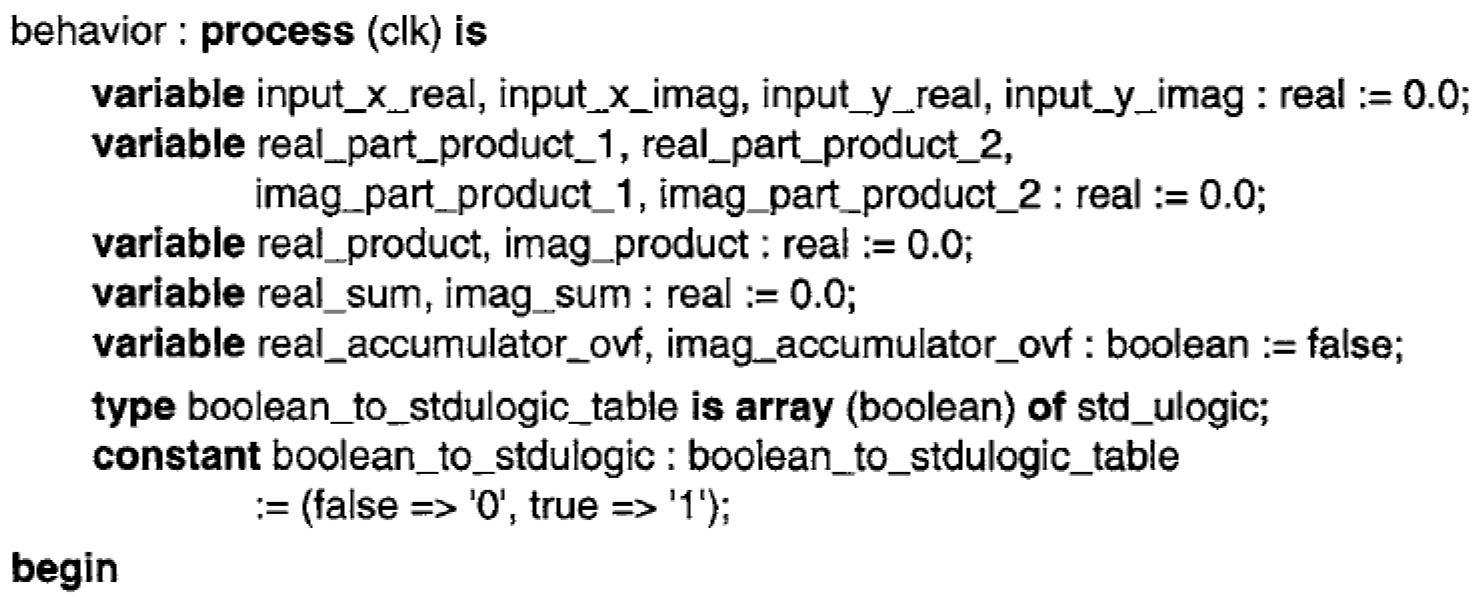
```
architecture behavioral of mac is
    constant Tpd_clk_out : time := 3 ns;
    signal fp_x_real, fp_x_imag,
           fp_y_real, fp_y_imag,
           fp_s_real, fp_s_imag : real := 0.0;
begin
    x_real_converter : entity work.to_fp(behavioral)
        port map ( x_real, fp_x_real );
    x_imag_converter : entity work.to_fp(behavioral)
        port map ( x_imag, fp_x_imag );
    y_real_converter : entity work.to_fp(behavioral)
        port map ( y_real, fp_y_real );
    y_imag_converter : entity work.to_fp(behavioral)
        port map ( y_imag, fp_y_imag );

    s_real_converter : entity work.to_vector(behavioral)
        port map ( fp_s_real, s_real );
    s_imag_converter : entity work.to_vector(behavioral)
        port map ( fp_s_imag, s_imag );
```

Convertor components are placed between the inputs and MAC structure to convert data

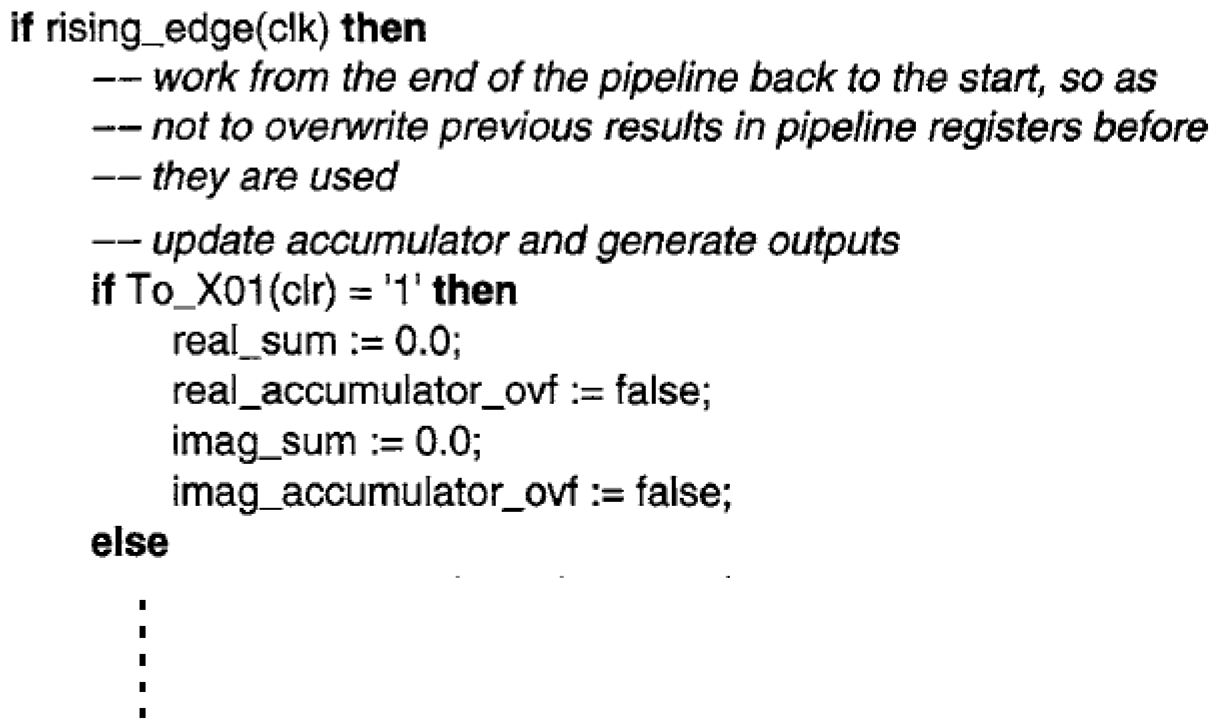
Direct instantiation—no component definition is needed

The main process



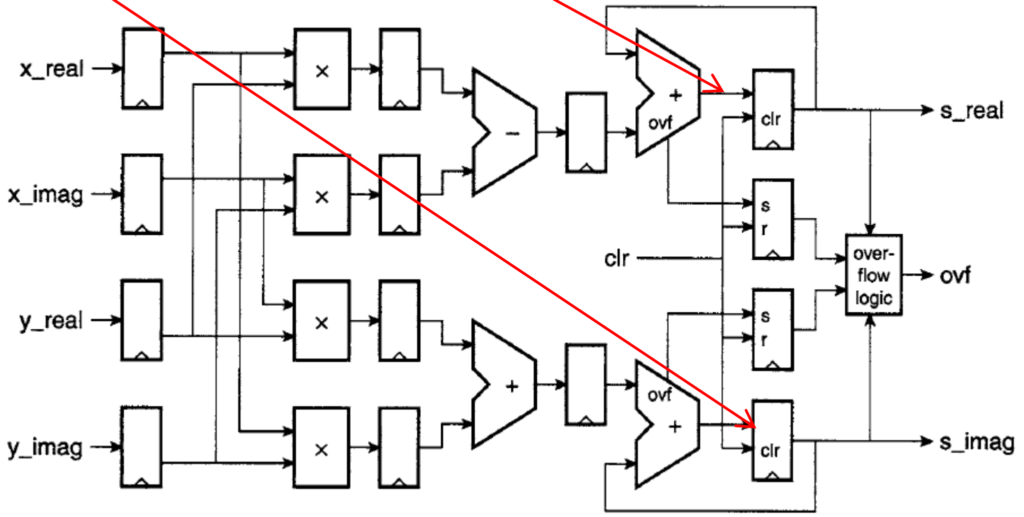
```
behavior : process (clk) is
    variable input_x_real, input_x_imag, input_y_real, input_y_imag : real := 0.0;
    variable real_part_product_1, real_part_product_2,
             imag_part_product_1, imag_part_product_2 : real := 0.0;
    variable real_product, imag_product : real := 0.0;
    variable real_sum, imag_sum : real := 0.0;
    variable real_accumulator_ovf, imag_accumulator_ovf : boolean := false;
    type boolean_to_stdulogic_table is array (boolean) of std_ulogic;
    constant boolean_to_stdulogic : boolean_to_stdulogic_table
             := (false => '0', true => '1');
begin
```

Main PROCESS- handling reset



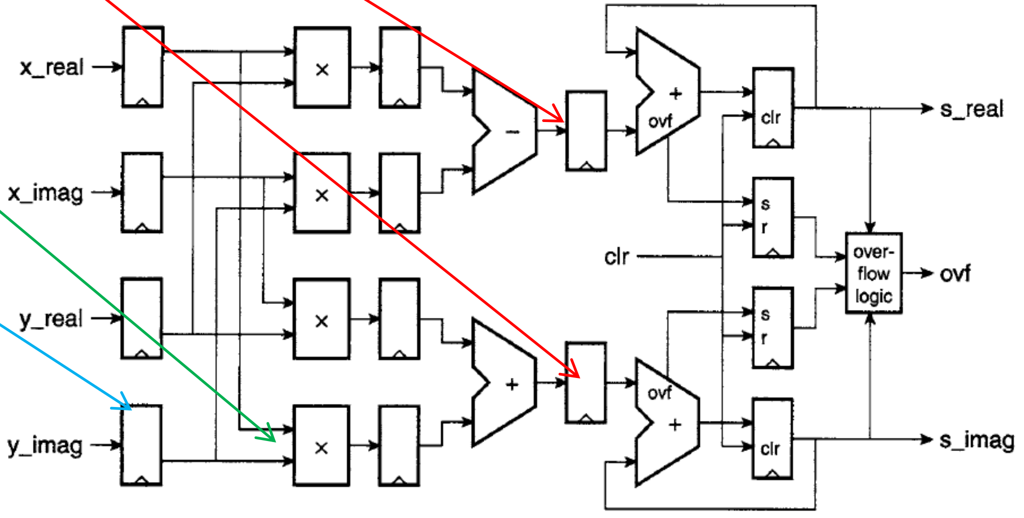
```
if rising_edge(clk) then
    -- work from the end of the pipeline back to the start, so as
    -- not to overwrite previous results in pipeline registers before
    -- they are used
    -- update accumulator and generate outputs
    if To_X01(clr) = '1' then
        real_sum := 0.0;
        real_accumulator_ovf := false;
        imag_sum := 0.0;
        imag_accumulator_ovf := false;
    else
      ⋮
```

```
behavior : process (clk) is
                  ⋮
         if rising_edge(clk) then
              if To_X01(clr) = '1' then
                 ⋮
              else
                   real_sum := real_product + real_sum;
                   real_accumulator_ovf := real_accumulator_ovf
                                           or real_sum < –16.0
                                           or real_sum >= +16.0;
                   imag_sum := imag_product + imag_sum;
                   imag_accumulator_ovf := imag_accumulator_ovf
                                           or imag_sum < –16.0
                                           or imag_sum >= +16.0;
              end if;
                 ⋮
```



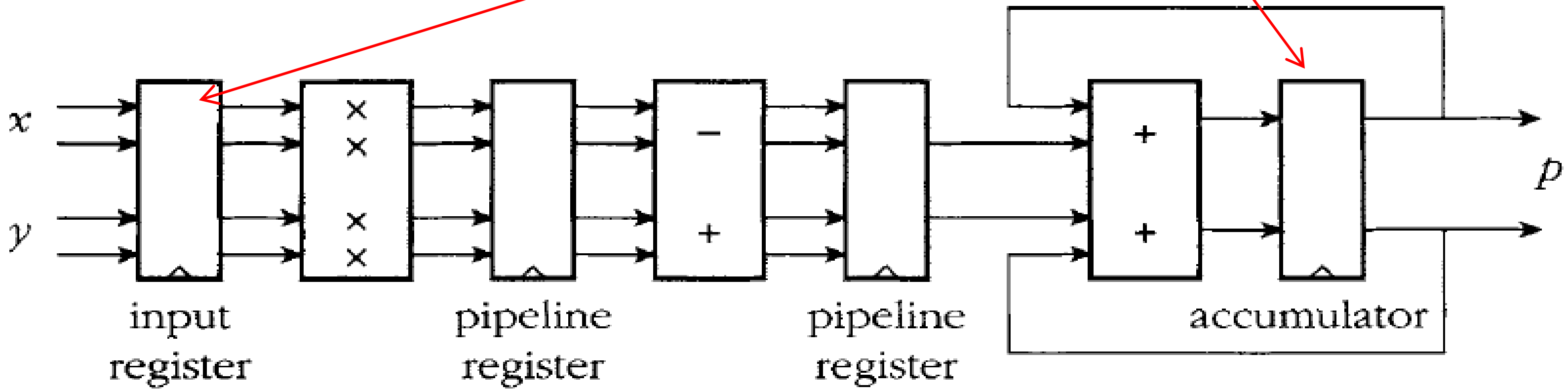
```
behavior : process (clk) is
                    ⋮
    -- update product registers using partial products
    real_product := real_part_product_1 – real_part_product_2;
    imag_product := imag_part_product_1 + imag_part_product_2;
    -- update partial product registers using latched inputs
    real_part_product_1 := input_x_real * input_y_real;
    real_part_product_2 := input_x_imag * input_y_imag;
    imag_part_product_1 := input_x_real * input_y_imag;
    imag_part_product_2 := input_x_imag * input_y_real;
    -- update input registers using MAC inputs
    input_x_real := fp_x_real;
    input_x_imag := fp_x_imag;
    input_y_real := fp_y_real;
    input_y_imag := fp_y_imag;

end process behavior;
```



```
        -- update product registers using partial products
        real_product := real_part_product_1 – real_part_product_2;
        imag_product := imag_part_product_1 + imag_part_product_2;
        -- update partial product registers using latched inputs
        real_part_product_1 := input_x_real * input_y_real;
        real_part_product_2 := input_x_imag * input_y_imag;
        imag_part_product_1 := input_x_real * input_y_imag;
        imag_part_product_2 := input_x_imag * input_y_real;
        -- update input registers using MAC inputs
        input_x_real := fp_x_real;
        input_x_imag := fp_x_imag;
        input_y_real := fp_y_real;
        input_y_imag := fp_y_imag;
    end if;
end process behavior;
```

Start from the last register and calculate the result (input of the next register) based on inputs.
Why starting from the last one? Why don't we start form the first register?



Generating final output

```
fp_s_real <= real_sum after Tpd_clk_out;
fp_s_imag <= imag_sum after Tpd_clk_out;
ovf <= boolean_to_stdulogic(
                    real_accumulator_ovf or imag_accumulator_ovf
                    or real_sum < –1.0 or real_sum >= +1.0
                    or imag_sum < –1.0 or imag_sum >= +1.0 )
            after Tpd_clk_out;
```

Reference

- See chapter 6 of "The designers guide to VHDL" for a complete design

Exercise

- Write a VHDL code to implement a pipelined datapath with highest throughput for *f*. Assume the latency of the addition, multiplication, and division operations are 12, 45, and 50 ms, respectively. The register latency is 2 *ms*. Use the *real* data type for the ports and predefined VHDL operations for *real* in the low-level entities.

$$f = ((a \times b + c \times d)/d)^2$$