Karlsruhe Institute of Technology

Institute for Information Processing Technologies

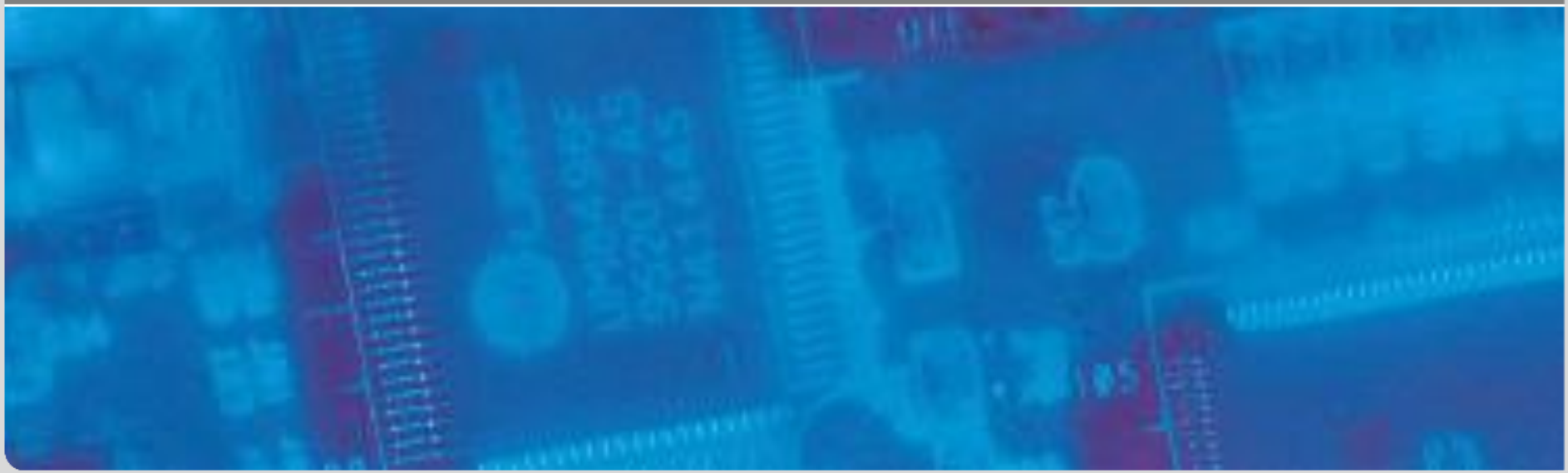
Head of institute
Prof. Dr.-Ing. Dr. h. c. Jürgen Becker
Prof. Dr.-Ing. Eric Sax
Prof. Dr. rer. nat. Wilhelm Stork

Digital Hardware Design Laboratory

Advanced VHDL and Pipelining

M. Sc. Simon Reder, M. Eng. Augusto Hoppe

Institute for Information Processing Technologies (ITIV)



VHDL ←→ Software Coding

- **VHDL describes Logic Resources**
  - Within Process: In Sequence (Process Update…)
  - Process-to-Process: In Parallel
- Preferable: **Synchronous Processes**
  - Easier implementation, more reliable translation into logic cells during Synthesis
- *Asynchronous Processes*:
  - Use as low logic as possible
  - Calculations may only depend on **synchronous** input data
  - **Use it with extreme care** → high potential of **Simulation/Runtime mismatches**
- On Process dependencies and latency issues…
  - Avoid using asynchronous „Handshaking“ or asynchronous inputs to Hard-IPs like BRAMs, DSP Slices etc.
  - Solve these issues using ***Pipelining***

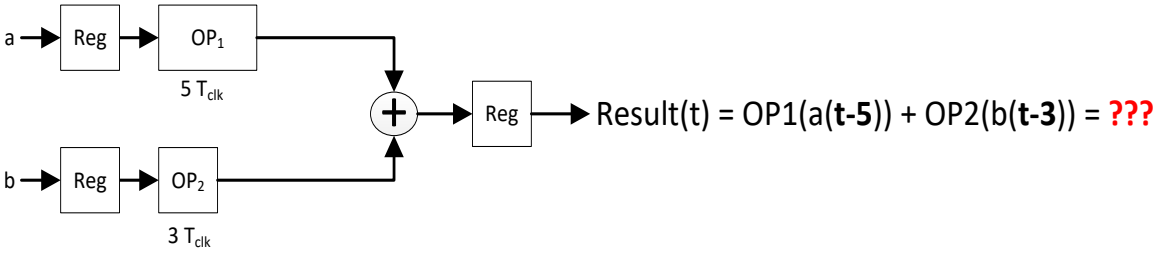
VHDL ←→ Software Coding

- Many operations known from C-programming **cannot or not efficiently** be implemented in Hardware Logic
  - **Delays** using given time values in seconds: „a <= b after 30ns;“
  - **Wait statements** – „wait until“ in modules for Synthesis is bad coding style
  - **Large storages** without using Hard-IPs (BRAMs):
    - „type my_array is array (natural range <>) of std_logic vector(31 downto 0);“
      „signal array : my_array (1023 downto 0);“
  - **Multiplication or Division**
    - by **non-power-of-two** values → long asynchronous logic paths
    - Division/Multiplication **within one Clock-Cycle**, like „c <= a / b;“
    - *Exception*: with DSP Slices c <= a*b for certain input sizes (amount is very limited on-chip) is possible but still a **very expensive operation**, thinking hardware-wise

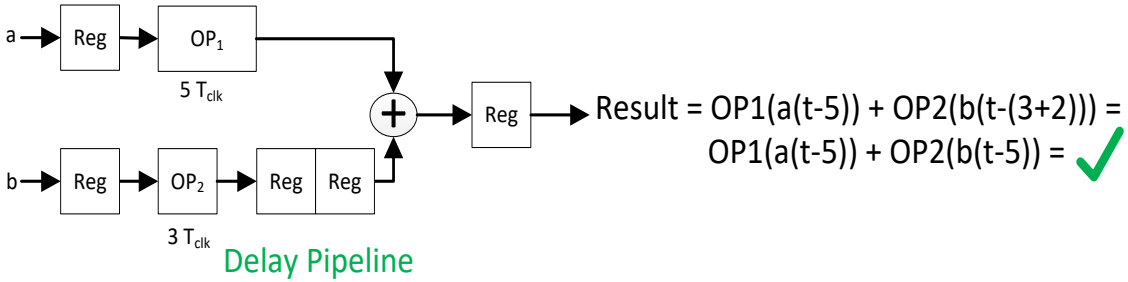
Pipelining in VHDL

- Reduces length of asynchronous paths → increases clock frequency
- Solves issues like:
  - Handshakes between processes requiring low latencies
  - Operations which cannot be performed within one clock cycle (→ e.g. Division)
- Assissts in compensating for ***different delays of processing Blocks***

Implementation with issues



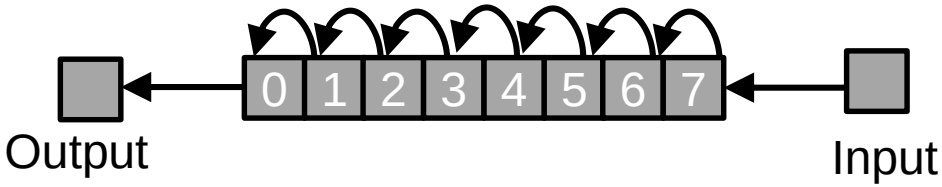
Solution



Pipelining in VHDL

- Description of Delay line in VHDL (Shift register):
  - Within Process: ***For*** loops

```
...
constant delay_const : integer := 8;
signal input_delayed : input_type(delay_const-1 downto 0);

begin
     process(clk, rstn, en)
     begin

        if rstn = '0' then
           input_delayed <= (others => '0');
        elsif rising_edge(clk) then
           input_delayed(delay_const-1) <= input;

          for i in delay_const-2 downto 0 loop
             input_delayed(i) <= input_delayed(i+1);
          end loop;

          output <= input_delayed(0);

...
```

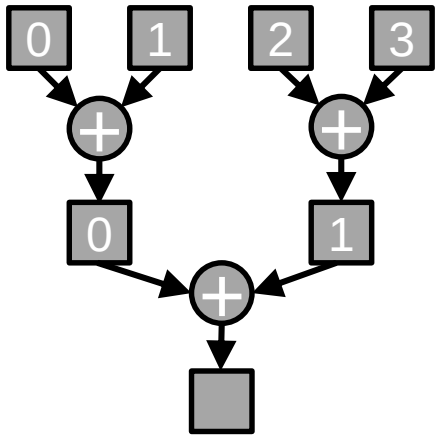


Pipelining in VHDL

- Similiar to for loops: ***Generate*** statement
  - Good for implementing regular structures with common properties, e.g. adder trees or in general for structural modelling.

```
Input_registering: process(clk)
      begin
        if rising_edge(clk) then
          if rst = '1' then
            stg_0_reg <= (others => (others =>'0'));
            stg_1_reg <= (others => (others =>'0'));
            stg_2_reg <= (others => (others =>'0'));
          else
            for i in 0 to 3 loop
              stg_0_reg(i) <= stg_0(i);
            end loop;

            for i in 1 to 0 loop
              stg_1_reg(i) <= stg_1(i);
            end loop;

            stg_2_reg <= stg_2;
          end if;
        end if;
      end process;
...
```

```
G0: for i in 0 to 3 generate
      stg_0(i) <= input(2*i) + input(2*i +1);
end generate;

G0: for i in 0 to 1 generate
      stg_1(i) <= stg_0_reg(2*i) + stg_0_reg(2*i +1);
end generate;

stg_2 <= stg_1_reg(0) + stg_1_reg(1);
```



Example (4 values)

Arithmetics in VHDL

- For arithmetic operations, the **size of the vector** (e.g. signed) of the ***result*** is usually of different size than the *input* vectors (here: ***VS = vector size***)

- Following rules apply for ***UNKNOWN*** values of a and b, for not losing information or to avoid values exceeding the vector size.

- **Addition / Substraction** (c = a+b): **VS(c) = max(VS(a), VS(b)) *+1***
- **Multiplication** (c = a*b): **VS(c) = VS(a) + VS(b)**

- *Simplifications may be possible*:
  If e.g. value b is known to be **< const_val**, then for **c = a*b** the vector size c may be chosen to **VS(c) = VS(a) + $\log_2$(**const_val**)**

Polynomial binary division

- Division **29 / 5 = 5, Remainder 4**
- Binary Division *(standard procedure)*

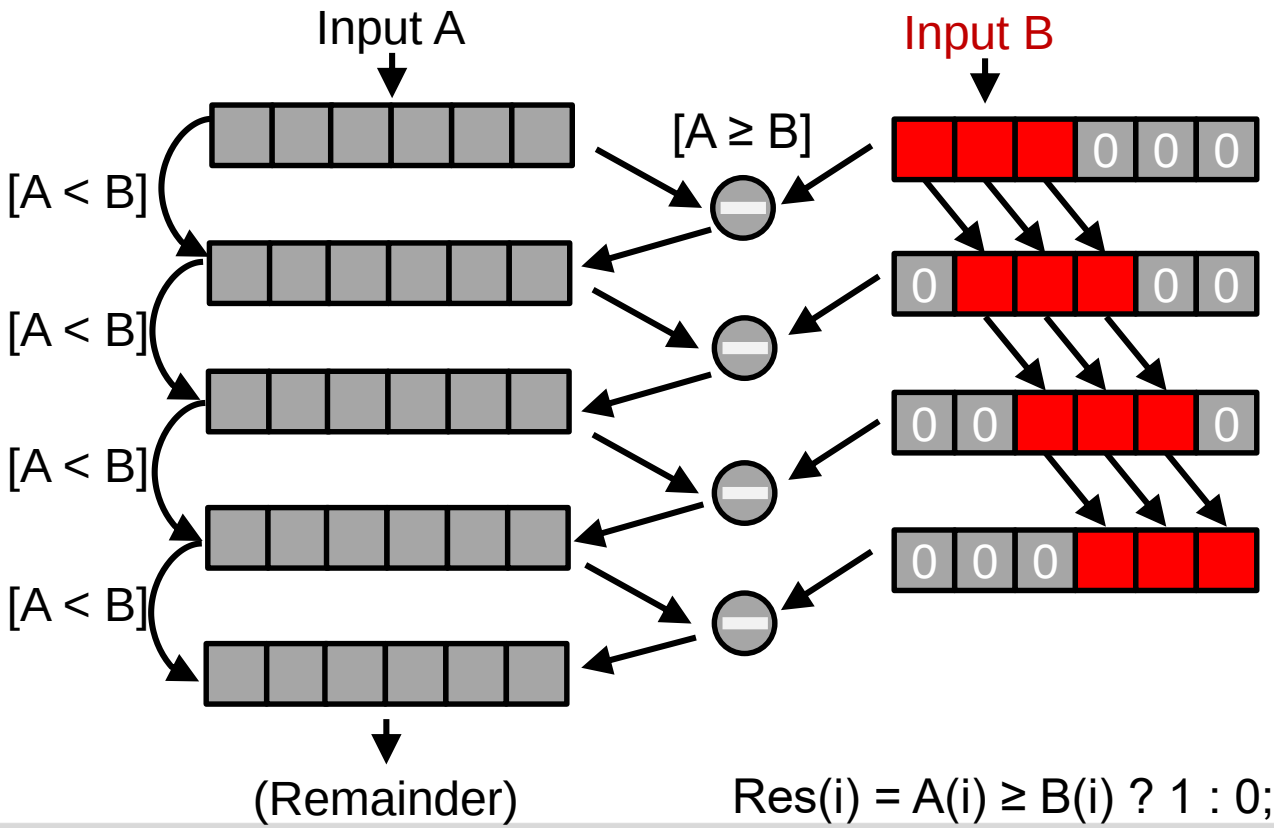
```
 011101 / 101 = 0101, Remainder 100
 011
-  0
 0111
- 101
  0100
-   0
   1001
-   101
    100
```

→ Efficient Hardware Implementation?

Polynomial binary division

- Division `29 / 5 = 5, Remainder 4`
- Binary Division (implementation with **shift registers**)

```
  011101 / 101 = 0101, Remainder 100
  011101
-(101000)
-      0
  011101
- 010100
   01001
- (01010)
-      0
   01001
-   0101
    100
```



Additional Tutorials/Videos on the Web

Interesting Tutorial Videos:

http://www.googoolia.com/wp/category/zynq-training/page/2/

Digilent learning platform:

https://learn.digilentinc.com/list?category=Digital

e.g.

- → **Zynq Hardware Architecture Highlights**
- → **Zynq Development Tools Overview**
- → **AXI Bus Overview**
- → **HDL Tutorials**
- → **…**

Not required for examination!