IN3160

Design principles and rules for FPGA and ASIC.

Application Specific Integrated Circuit

```
                    ┌──────────┐
                    │   ASICs   │
                    └──────────┘
          ┌──────────┬────┴────┬──────────┐
          ▼          ▼         ▼          ▼
  ┌────────────┐ ┌────────────┐ ┌────────────┐ ┌────────────┐
  │ Gate Arrays│ │ Structured │ │  Standard  │ │Full Custom │
  │            │ │   ASICs    │ │    Cell    │ │            │
  └────────────┘ └────────────┘ └────────────┘ └────────────┘

              ───────────────────────────►
                  Increasing complexity
```
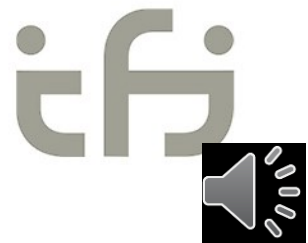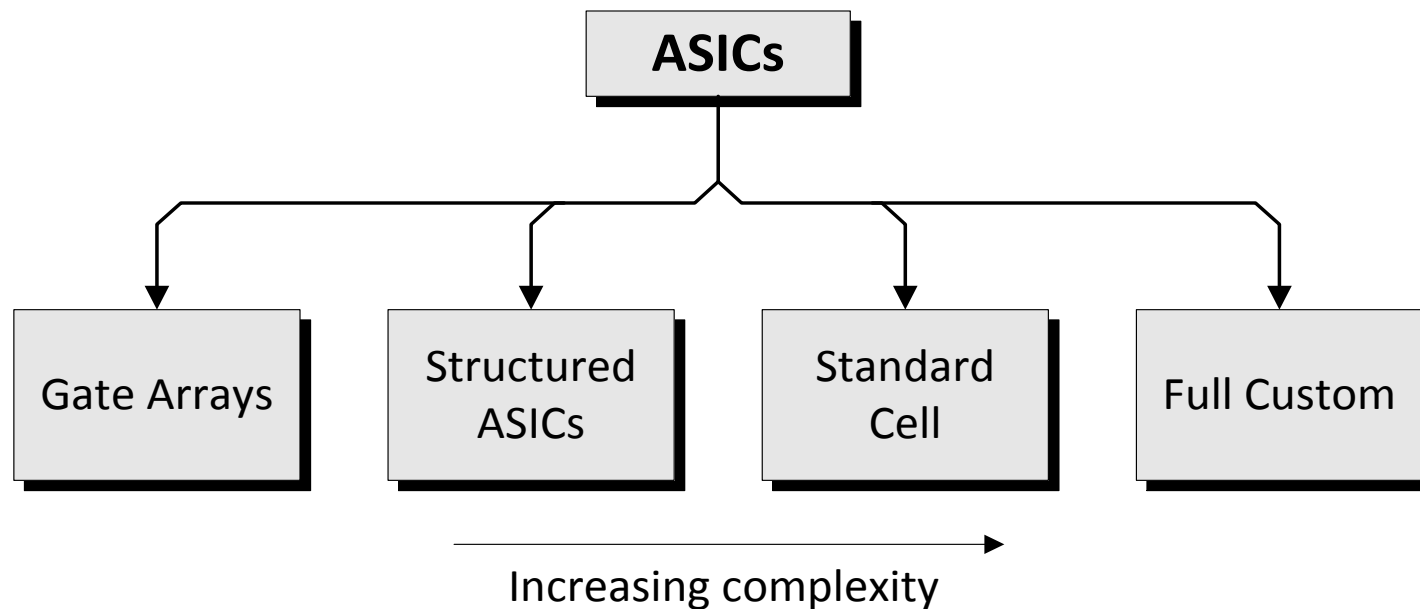
See: https://en.wikipedia.org/wiki/Application-specific_integrated_circuit

When should an FPGA/CPLD be used?

- First choice for design with the following exceptions:
  - Extreme performance requirements (high clock frequency)
  - Will be produced in massive quantities
  - Very complex design (very large FPGAs are expensive)
  - Analog electronics needed on the same chip
  - Designs where low power is critical (mobile applications)
- ASIC design is prototyped on FPGAs and conversion to ASIC is outsourced if the company lacks dedicated ASIC designers
- ASIC vs FPGA projects (2003):
  - 1500-4000 new ASIC projects each year
  - 450 000 new FPGA projects each year

Main advantages of FPGA development over ASIC

- Lower economical risk with a much lower initial investment (**zero** <u>N</u>on<u>R</u>ecurring <u>E</u>ngineering (NRE) cost)

- Shorter development time due to easier re-programming. Results in faster time-to-market

- SRAM and flash based can be re-programmed both during development and in-system after delivery to the customer

FPGA-to-ASIC

- One or more FPGAs are used for prototyping an ASIC design

- A challenge that ASIC does not have the same blocks as the FPGA:

  - A library can be made of functions (multipliers, memory blocks, etc.) that exists in an FPGA to allow use of this in ASIC, though this limits the ASIC synthesis

- The RTL code should be the same for both FPGA and ASIC

- Example: ON Semiconductor and Intel (www.easic.com) offers implementation of FPGA based solutions in ASICs for higher performance, lower production price and lower power consumption.

ASIC-to-FPGA

- Original ASIC is out of production.

- Expand functionality of an ASIC without a new large investment; i.e. FPGA development is much cheaper than ASIC development.

- The size of modern FPGAs have made it possible to implement ASICs made a few years ago in <u>a single</u> FPGA chip.

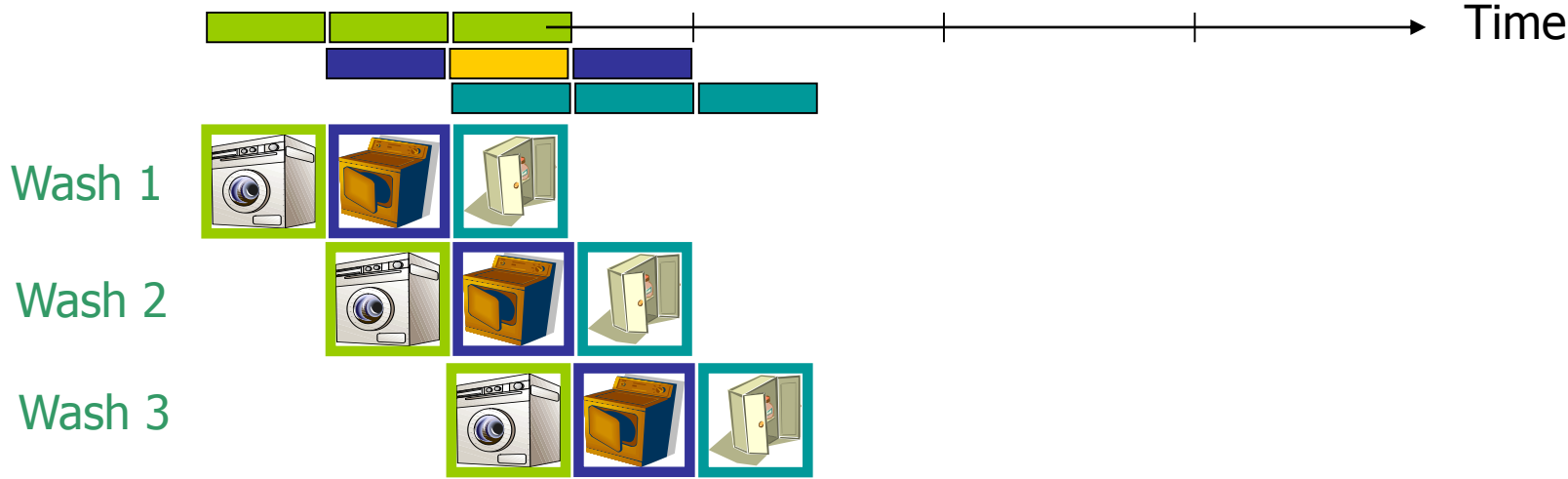- Requires updating the ASIC design to adapt to FPGA specific functionality and features; i.e. "archeology" ☺
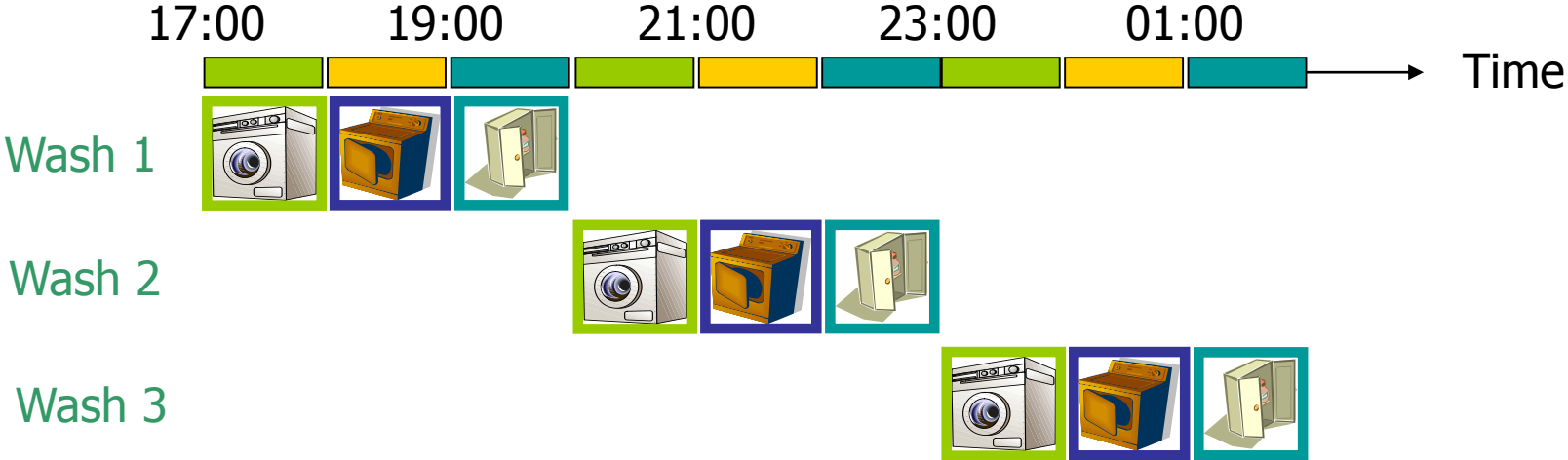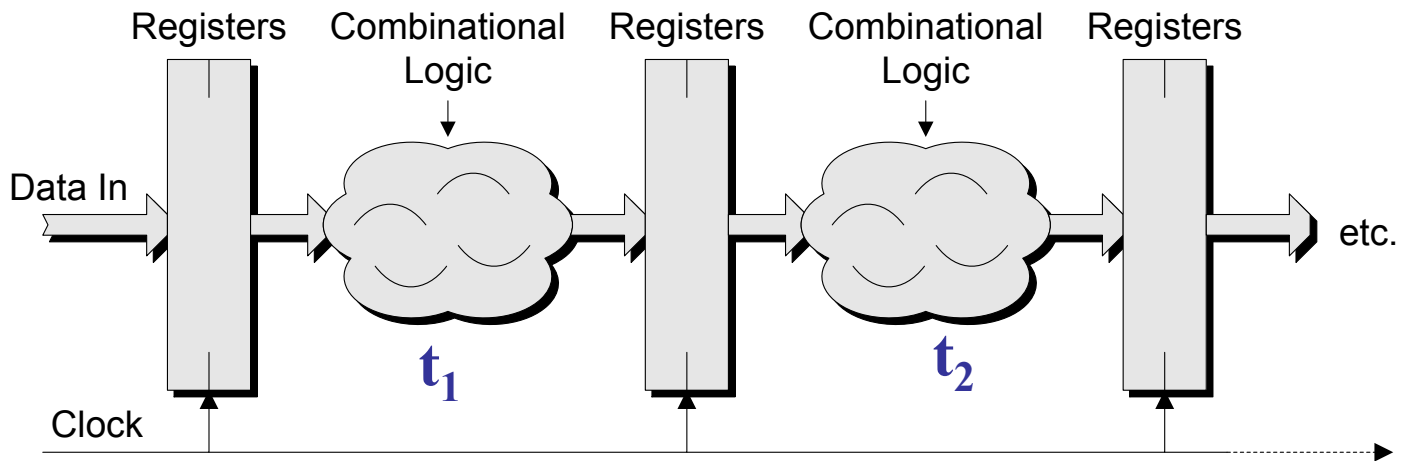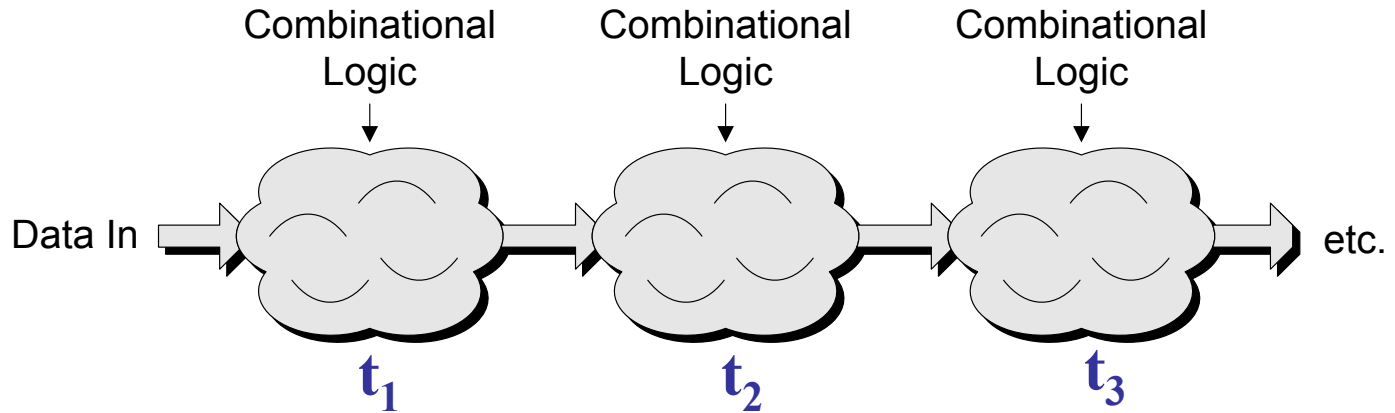
Coding styles

- Pipelining

- Number of logic levels

- Asynchronous logic

- Use of clocks

- Lathes and registers

Pipelining

Pipelining in digital systems

Combinational Logic   Combinational Logic   Combinational Logic

Data In → [cloud] → [cloud] → [cloud] → etc.

t_1 t_2 t_3

Registers   Combinational Logic   Registers   Combinational Logic   Registers

Data In → [register] → [cloud] → [register] → [cloud] → [register] → etc.

t_1 t_2

Clock

Pipelining with VHDL example

In the module compute shown below, the sum of the numbers a, b, c and d shall be calculated as 16 bits. The output result with16 bits is set to max value equal to x"FFFF" (i.e. all bits set to '1') when the sum is greater than x"FFFF" and the signal max shall be set to '1' at the same time.

```vhdl
entity compute is
  port
    (rst    : in  std_logic;
     clk    : in  std_logic;
     a      : in  std_logic_vector(15 downto 0);
     b      : in  std_logic_vector(15 downto 0);
     c      : in  std_logic_vector(15 downto 0);
     d      : in  std_logic_vector(15 downto 0);
     result : out std_logic_vector(15 downto 0);
     max    : out std_logic);
end entity compute;
```

Pipelining with VHDL example cont.

```vhdl
architecture rtl of compute is
begin
  process (rst, clk) is
    variable result_i : unsigned(17 downto 0);
  begin
    if rst = '1' then
      result <= (others => '0');
      max    <= '0';
    elsif rising_edge(clk) then

      result_i := unsigned("00" & a) + unsigned("00" & b) +
                  unsigned("00" & c) + unsigned("00" & d);

      if result_i > "001111111111111111" then
        result <= (others => '1');
        max    <= '1';
      else
        result <= std_logic_vector(result_i(15 downto 0));
        max    <= '0';
      end if;

    end if;
  end process;
```

Pipelining with VHDL example problem

It turns out that there are timing errors during implementation in the selected technology and with the selected clock frequency.

The architecture *rtl* has to be changed to a new architecture *pipelined_rtl* that have maximum 1 add operation (i.e. + operator) and 1 comparison operation (i.e. the statement result_i > "001111111111111111") in <u>one clock period</u> to achieve the timing requirement (i.e. clock frequency).

Multiple add and comparison operations can still be performed in parallel in each clock period (i.e. many adder and comparator modules available in the FPGA hardware)

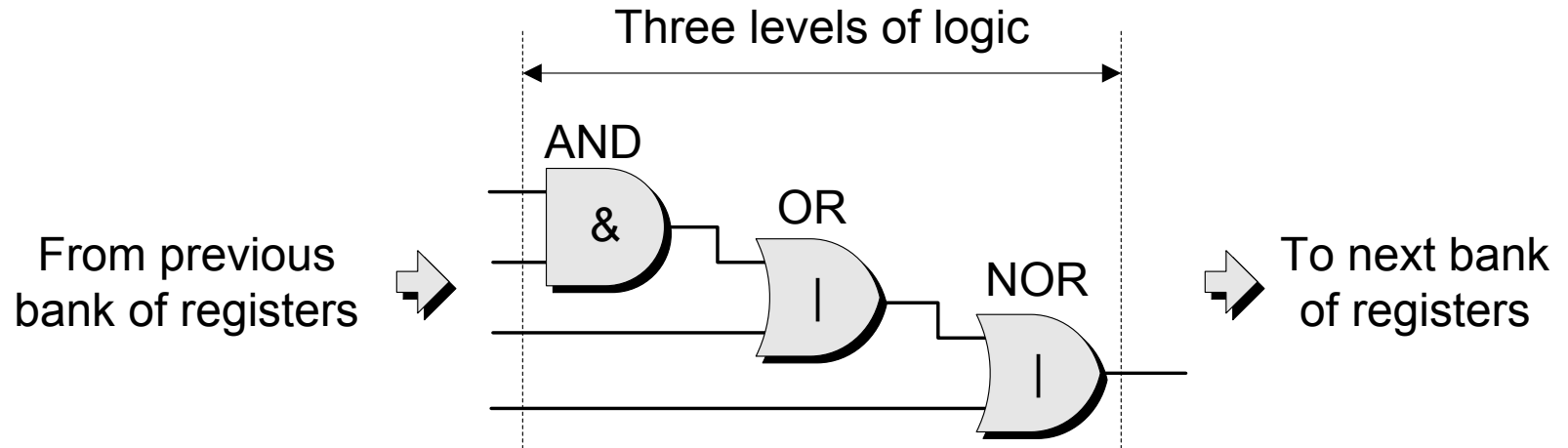# Pipelining with VHDL; solution with pipelining

```vhdl
architecture pipelined_rtl of compute is
  signal ab_tmp    : unsigned(16 downto 0);
  signal cd_tmp    : unsigned(16 downto 0);
begin
  process (rst, clk) is
    variable result_i : unsigned(17 downto 0);
  begin
    if rst = '1' then
      ab_tmp    <= (others => '0');
      cd_tmp    <= (others => '0');
      result    <= (others => '0');
      max       <= '0';
    elsif rising_edge(clk) then

      ab_tmp    <= unsigned('0' & a) + unsigned('0' & b); -- NOTE: signal assignment
      cd_tmp    <= unsigned('0' & c) + unsigned('0' & d); -- NOTE: signal assignment

      result_i := ('0' & ab_tmp) + ('0' & cd_tmp); -- NOTE: variable assignment
      if result_i>"001111111111111111" then
        result <= (others => '1');
        max    <= '1';
      else
        result <= std_logic_vector(result_i(15 downto 0));
        max    <= '0';
      end if;
    end if;
  end process;
end architecture pipelined_rtl;
```

Number of logic levels

Three levels of logic

AND

&

OR

|

NOR

|

From previous bank of registers

To next bank of registers

- The number of logic levels are more critical in FPGAs where the delay between ports often are higher than in ASICs.
- FPGA designs may use more pipelining than ASIC designs to achieve the required clock frequency since each logic cell in a FPGA contains both a LUT <u>and a register</u>, but it will increase power consumption.
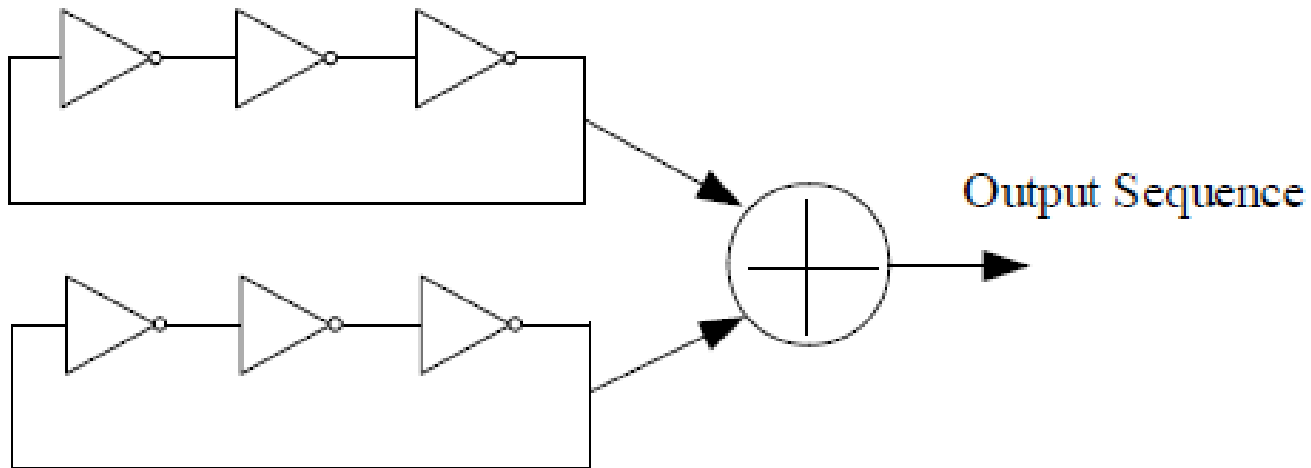
Asynchronous logic
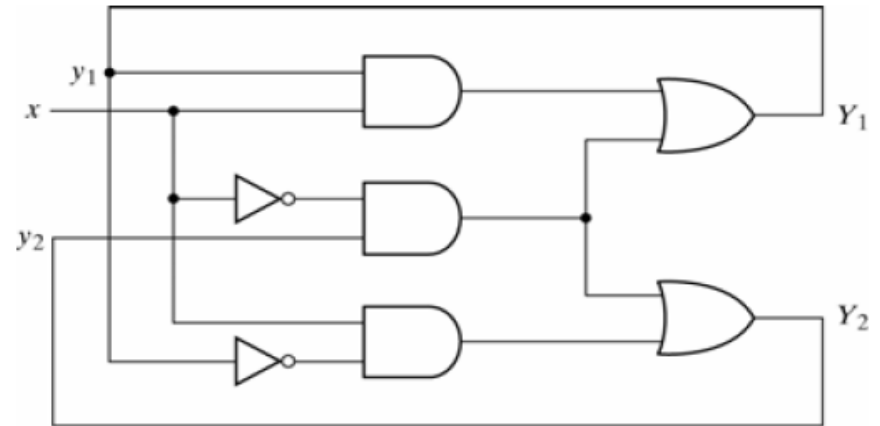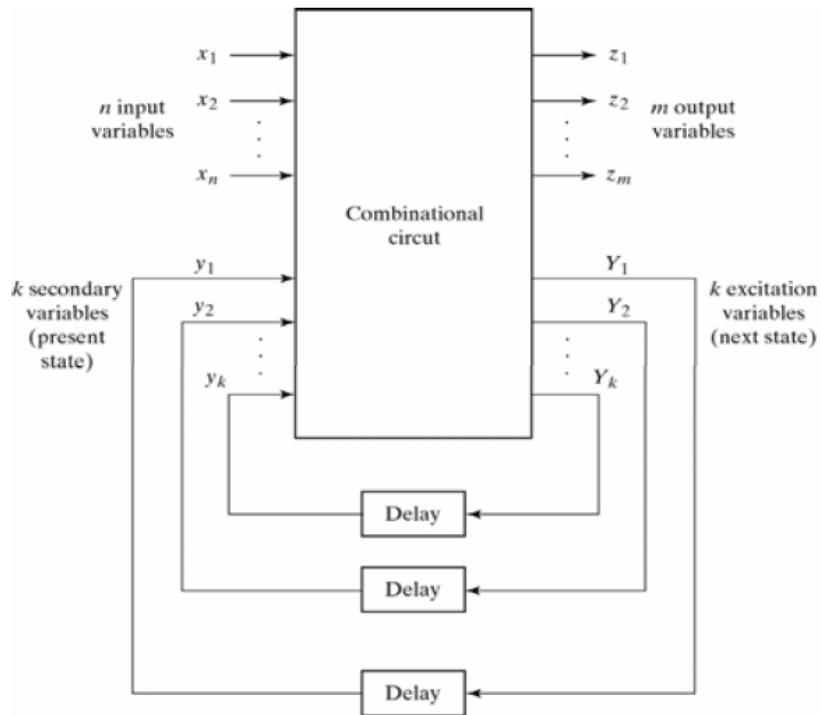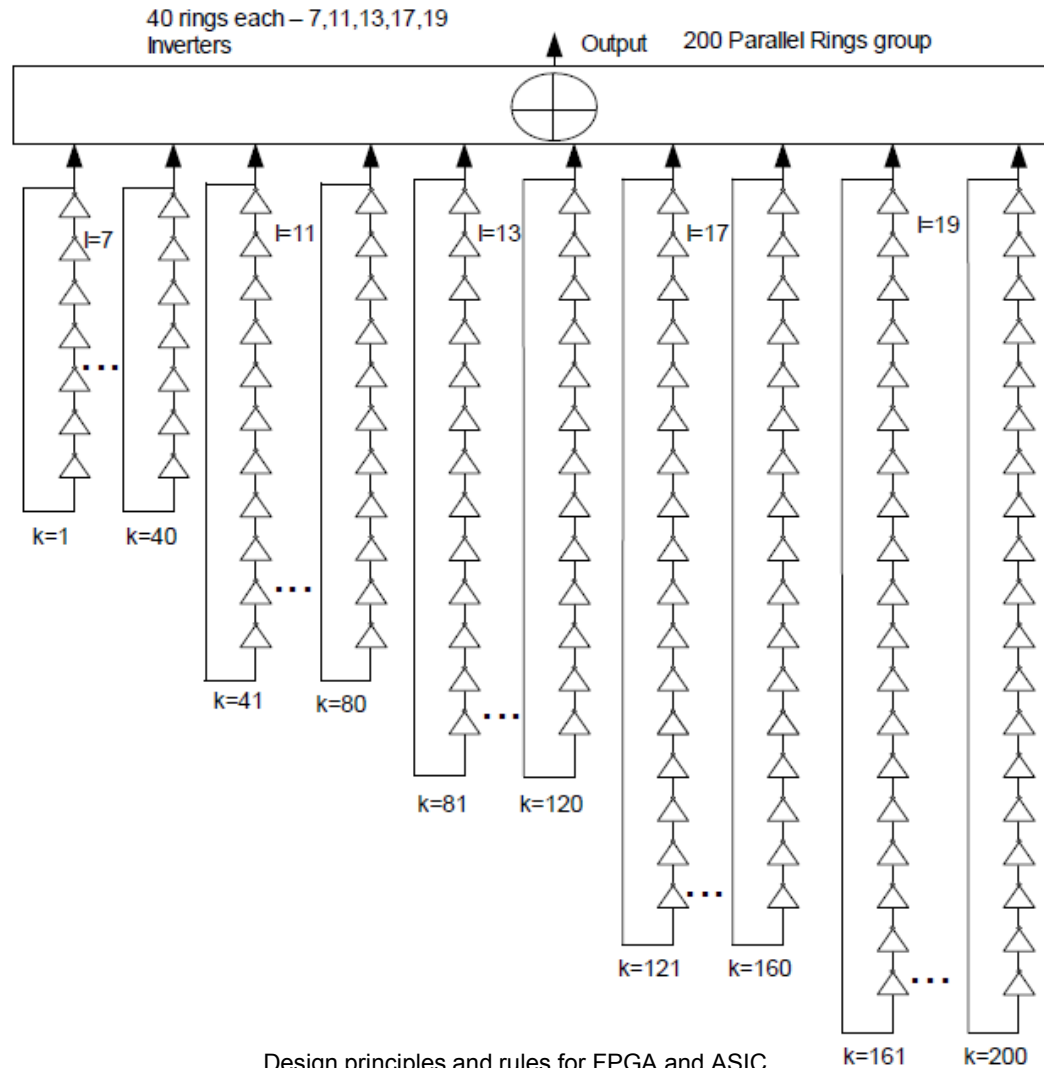
Asynchronous design principles

- FPGAs can usually not have asynchronous designs (in contrast to ASICs), because the behavior would change each time place and route is performed.

- FPGAs shall usually always use registers in feedback loops.

- Delay chains of combinatorial elements are hard to make predictable in FPGAs

- Asynchronous logic design in FPGA is used in special cases as in a True Random Number Generator (TRNG) module (*ref. P.S.Sundaram, 2010*).

Asynchronous TRNG basic principle with XOR'd output

Output Sequence

Asynchronous TRNG multiple ring design

40 rings each – 7,11,13,17,19 Inverters   Output   200 Parallel Rings group

I=7   I=11   I=13   I=17   I=19

k=1   k=40
k=41   k=80
k=81   k=120
k=121   k=160
k=161   k=200

Clocks

- Limited number of clock distribution networks in FPGAs limit the number of clock domains

- General FPGA inputs can often not be used for clock signals. Check vendor specifications!

- FPGA designers do not need to fine tune clock paths. The place and route tool for FPGA automatically does this (hurrah ☺)

- Clock enabling, and not clock gating, should be used in FPGAs

  - Clock gating can be done, but only using special clock gating cells (for example with the "BUFGCE" clock buffer from Xilinx)
  - Due to limited number of clock gating cells (e.g. BUFGCE) should clock gating be used for clocks to one or more modules.
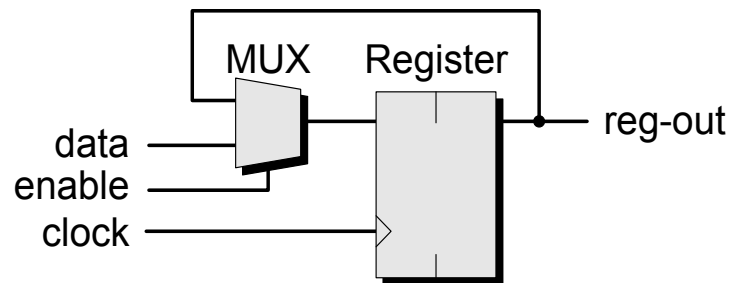
Clock gating and clock enabling
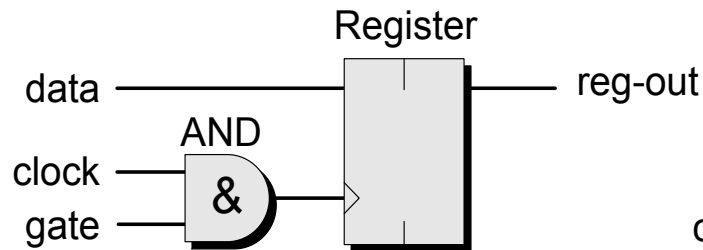
(a) Clock gating principle
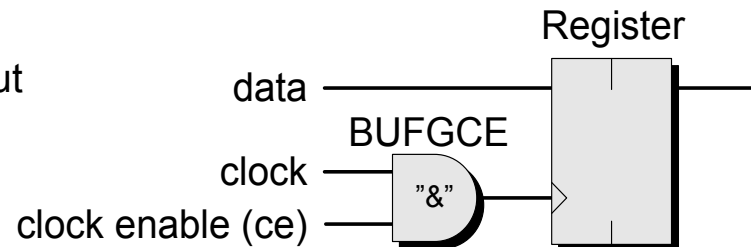
(b) Safe clock gating with Xilinx BUFGCE clock buffer
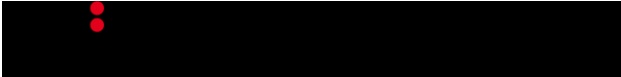
(c) Clock enabling ("then")

(d) Clock enabling ("now")

Design principles and rules for FPGA and ASIC
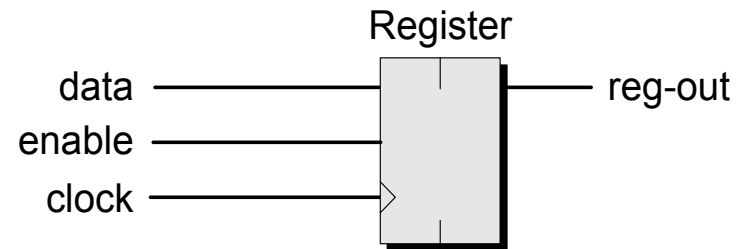
Clock generation, latches and synchronous vs. asynchronous reset

- Clock generating modules (DCM / MMCM / PLL) and clock distribution network already implemented

- Latches

  - Do not use latches in FPGA (is mostly true for ASIC as well)

    - **Very good example at: www.doulos.com/knowhow/vhdl_designers_guide/synthesising_latches**

- Some FPGA chips support both asynchronous and synchronous set and reset of registers, while ASIC and some FPGA chips only support asynchronous set and reset.

Implementation on FPGA

- ## Choice of FPGA and resource sharing

  – The choice of FPGA device should be done with the goal of using most of the included functionality, given the **"using it or loosing it"** principle

  – It is often more power efficient to use independent functional units than using resource sharing based on multiplexers, as long as enough functional units such as multipliers are available

- ## Finite State Machines (FSM) state encoding

  – "One-hot" encoding may instead of binary encoding be a space- and timing-efficient technique due to the high number of registers in an FPGA, though it is not as often used since multiple states can be present at the same time (not a "safe" FSM)

- ## FPGAs comes production tested from the vendor (e.g. Xilinx, Intel, Microchip, Lattice) (hurrah ☺)

  – ASICs must be Designed For production Test (DFT), which requires logic resources, design time and tools for generating production tests.

General C (C++) with High Level Synthesis (HLS)
HLS is lectured in IN5200 and used in lab ☺

- Matlab can also generate VHDL/Verilog synthesisable code and C-code for functional verification (i.e. DPI-C for SystemVerilog).

User interaction and guidance

Pure C/C++ → **Pure C/C++ Synthesis**

FPGA target

Verilog / VHDL RTL → **RTL Synthesis** → **LUT/CLB-level netlist**

Auto-generated, implementation-specific

ASIC target

Verilog / VHDL RTL → **RTL Synthesis** → **Gate-level netlist**

- Non-implementation-specific
- Easy to create
- Fast to simulate
- Easy to modify
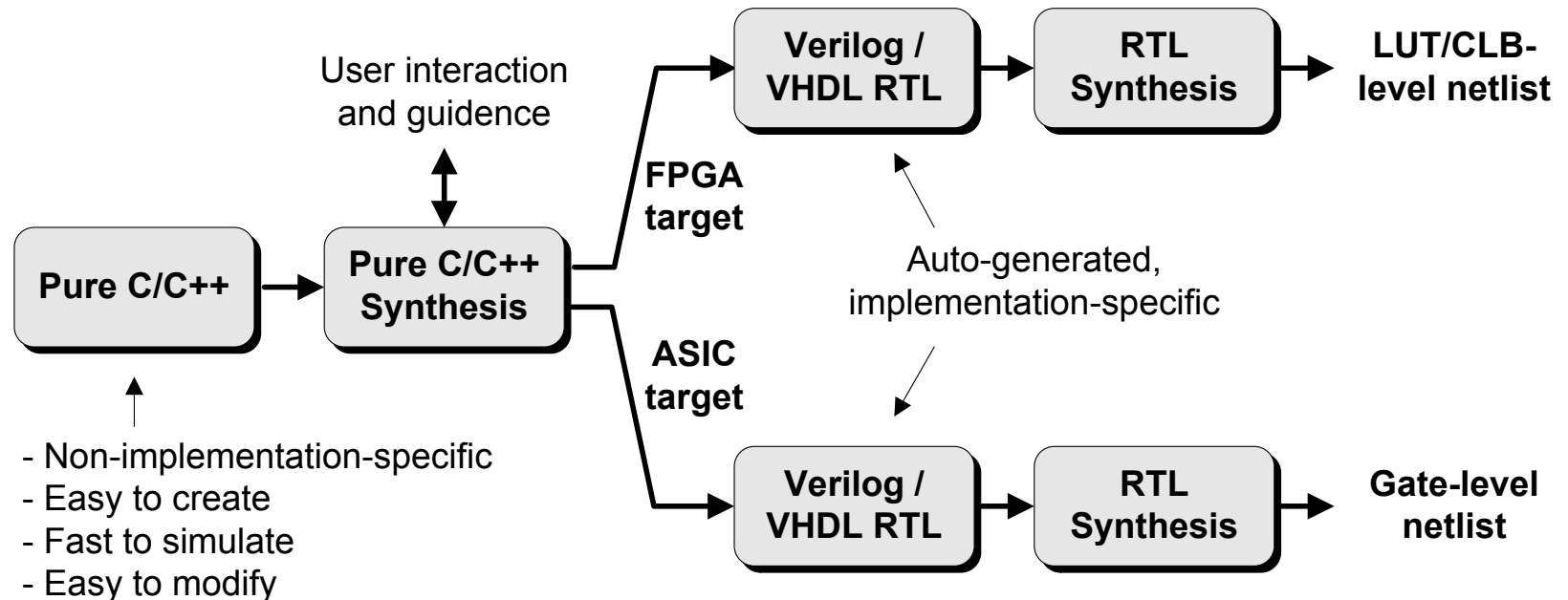
FPGA and ASIC development process

- Most companies have a separate development process for FPGA and ASIC.

- Includes design rules, requirements, milestones, documentation requirements, and responsibilities for project members.

- Design reviews and tools are used to reduce risk of errors and miscommunication

Code rules I

- Only a single statement is allowed per line

- Use (if possible) only active high signals (i.e. value '1'). External signals that are active low shall be inverted in the first entered module. Exceptions are active low reset signal and signals to IPs.

- Avoid internal tristate busses.

- Allowed types:
  - **std_logic**
  - **std_logic_vector** - only used for busses that are not numbers
  - **unsigned** - used for all unsigned numbers
  - **signed** - used for all signed numbers
  - **integer** – <u>**shall**</u> be avoided if possible
  - **enumerated types** - can be used for state machine variables. If used in several modules, they shall be defined in a common project package.
  - **boolean** – used for boolean operations (std_logic is preferred)
  - **composite types** – collection of above types. **Records can be used for grouping signals** (e.g. cpubus = data + control).

- **Only explicit port mapping is allowed.**

Code rules II

- Vectors shall be defined as **MSB down to LSB**, e.g. std_logic_vector(**7 downto 0**). LSB shall always be bit 0 if there are no other special reasons.

- An original signal type shall be used throughout the hierarchy if the target port is of the same type. Signals from/to the core (or higher top level) module should be of the type std_logic and std_logic_vector.

- Port ordering in entity shall be: resets, clocks, common signals, signals grouped by functionality or module. May group signals by each interface alphabetically, inputs first, outputs and then I/O.

- Allowed packages:
    - ieee.std_logic_1164.all
    - ieee.numeric_std.all
    - ieee.std_logic_textio
    - std_textio
    - std_developerskit
    - project and company packages
    - UVM and UVVM testbench packages

Code rules III

- Concurrent statements should only be used for assigning the outgoing port to its internal signal (e.g. **res <= res_i**) and for creation of tristate busses on top level.

- Do not use too many, or too few processes.

- Finite state machines can be described either in two processes (one sequential and one combinatorial) or just as a single process (more about this in IN5200).

- It is recommended to use functions **rising_edge** and **falling_edge** instead of **event** when describing clock edges.

- Variables can be used both for internal process calculations and for register inferring. If the variable is only used for intermediate calculation, always assign the variable before it is used to avoid latches.

- A multi level if-else statement shall only be used when a priority encoder is intended. Otherwise case statement shall be used. Always use default value if latch is not intended. Default values can also be set first in the process, above if/case statement. In a case statement, use others (do not use null).

Code rules IV

- Use parenthesis to group expressions in IF-statements to improve readability.

- Avoid purely combinational modules as they are not recommended for synthesis. If possible, all output signals of a module shall come from registers.

- Asynchronous signals or signals crossing clock domain boundaries shall be synchronized to avoid meta-stability. Asynchronous input signals shall be synchronized in the first entered module.

- Use tabs automatically substituted with spaces when writing code. Indentation shall be 2 or 3 spaces.

- Longer concept description comments for the module shall be part of the header or placed early in the file. Shorter line comments shall be used for each process or functional part of the code.

- A comment declaration of each port and signal (each on a separate line) shall be used. Comments shall be placed above or to the right of the code. Align comments if possible.

Naming rules I

- Upper case shall only be used for: **constants**, **enumerated type literals**, **generics** and **process labels**. Lower case shall be used for remaining names including file names. All names shall be as short as possible, but always meaningful.

- Module names
  - Use short module names.
  - Do not use underscore in module names (except when prefix used).
  - Preferably 2 to 5 characters

- Instance names:
  - Always use instance labels ending with _? (e.g. module mod_reg instantiated with label mod_reg_0, mod_reg_1).

- Design units (e.g. entity, architecture) may be in separate files.

Naming rules II

- Predefined architecture types:
    - str   structural
    - rtl   register transfer level
    - beh   behavioural (i.e. <u>not</u> synthesizable)
    - dmy   dummy (empty). All outputs set to <u>inactive</u> values.

- Avoid mixing architecture types (e.g. rtl and str).

- File and design unit naming examples:
    - Entity: uart_ent.vhd
    - Architecture: uart_rtl.vhd, uart_str.vhd, uart_dmy.vhd
    - Configuration: uart_cfg.vhd
    - Package def.:   nova_pck.vhd
    - Package body: nova_bdy.vhd
    - Testbench: tb_uart.vhd or t_uart_vhd
    - Testbench config: tb_uart_cfg.vhd or t_uart_cfg.vhd

Naming rules III

- Predefined suffixes for signals
  - _n   negative polarity; active low
  - _i   internal signal of outgoing port
  - _d1, _d2   delayed signal (i.e. number of cycles).
  - _s1, _s2   synchronized signal (i.e. number of cycles).
  - _str   strobe signal (i.e. one clock cycle long)

- Predifined names
  - Clock and reset signals shall be preserved throughout the hierarchy.
  - Clocks: default clock signal shall be mclk. If other clocks exist, the name shall be clk_? and include frequency (m=MHz and k=kHz), e.g. clk_34k, clk_10m, clk_10m24
  - Resets: rst, rst_n
  - Interrupts: signal names shall be "irq_" (e.g. irq_fifo_empty).
  - Process labels shall start with prefix P_ (e.g. P_DATA_READ:)
  - Generate labels shall start with prefix G_ (e.g. G_MUX:)

Development parallelism and pipelining

- Designs are usually divided into a control structure and one or more data paths
- The control structure is often implemented first with:
  - Processor interfaces (e.g. AXI4, PCI-E)
  - Access to control registers and RAMs.
  - Extra functionality for testing interrupts to processor from register modules.
  - Communication between internal and external processors
  - Complete test circuit with top module and internal modules like core, CRU, etc.
    - **No detour or dead end!**
  - Dummy modules where all inputs may be connected to all outputs or all output signals set to inactive '0' or '1'.
  - SW has to generate lab test programs for register access, RAM access and interrupt testing
    - **Test basic infrastructure before functional testing!**
- Data path and data path control
  - Full or often incremental release of data paths for target/lab verification with SW
  - Changes in the initial version of the control structure modules may be needed

Simulation vs lab debug

- It is very important to have a thoroughly simulated design with good test benches before lab. testing.

- Most of the warnings from simulation, synthesis and P&R tools should be removed or explained.

- Use on-chip logic analyzer (like Xilinx ILA, Synopsys Identify or Intel/Altera's Signal tap) to find internal FPGA bugs, but also use it to find errors or misunderstandings in external component interfaces and timing.

- Errors in design should be identified by simulation and then fixed and simulated before more lab. testing.

    – **Do not hope/believe the functional error is fixed – know that it is fixed!**

- Simulation environment should be used actively during lab/system testing