ELEC 305

Digital System Design Lab

Fall 2024

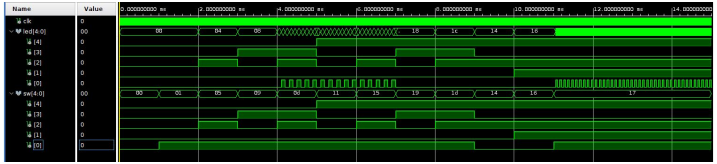
Lecture 4:

Performance Analysis and Optimization

Recap

- Up to now we've seen how to…
  - **describe** a circuit using ("DUT") VHDL based on a given set of specifications
  - use Vivado to automatically **synthesize** the DUT and **implement** it on the FPGA
  - use VHDL to generate test signals for DUT and **simulate** its **behavior** to characterize its accuracy
- However, we don't know how to characterize circuit timing performance, and fix it if it's not satisfactory.
- Today we'll see analyses and related optimizations to improve timing performance.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity coffeemaker is
  Port ( clk : in  STD_LOGIC;
         led : out STD_LOGIC;
         sw  : in STD_LOGIC
       );
end coffeemaker;

architecture Behavioral of coffeemaker is
  signal pulse : std_logic := '0';
  signal count : integer range 0 to 199999999 := 0;
begin
  process(clk, sw)
  begin
     ...
  end process;

  led <= pulse;
end Behavioral;
```



Recap

- With analyses and optimizations, we will have covered the whole digital design workflow

- This will conclude Part 1 of the course, and we'll use these skills in Part 2 while building useful algorithms on FPGAs

1) ***Create Project***: Launch the IDE and answer some basic questions about your FPGA project.
2) ***Add HDL***: Use the code editor tool to enter your own kick-ass HDL.
3) ***Add IP Modules***: Intellectual Property (IP) modules are HDL written by the FPGA vendor. Some are free.
4) ***Run Simulation***: Use the simulation tool to test that your HDL is logically correct.
5) ***Add Constraints***: Use Tcl language to write constraints (eg. matchup between HDL-signals and FPGA-pins).
6) ***Run Synthesis***: Create a digital circuit representation for your HDL.
7) ***Run Implementation***: Run timing analysis and place/route the digital circuit from synthesis into the FPGA.
8) ***Check, Correct, Iterate***: Check that design has passed timing analysis. Correct HDL and rerun tools as necessary.
9) ***Generate Bitstream***: Create a file the describes the results of implementation (ie. the FPGA configuration).
10) ***Program the FPGA***: Configure the FPGA hardware by downloading the bitstream into the FPGA.

e.g., our debouncer

.xcd

this is the new part

img src: [“FPGAs with VHDL: first steps”, Helen DeBlumont]

Outline

- Performance attributes in digital circuits
- Quantifying timing performance
- Static timing analysis, its differences with simulation, and why we care
- Optimizations: RTL-level → pipelining, parallelization and others
- Optimizations: Using primitives (e.g., DSP cores)

Outline

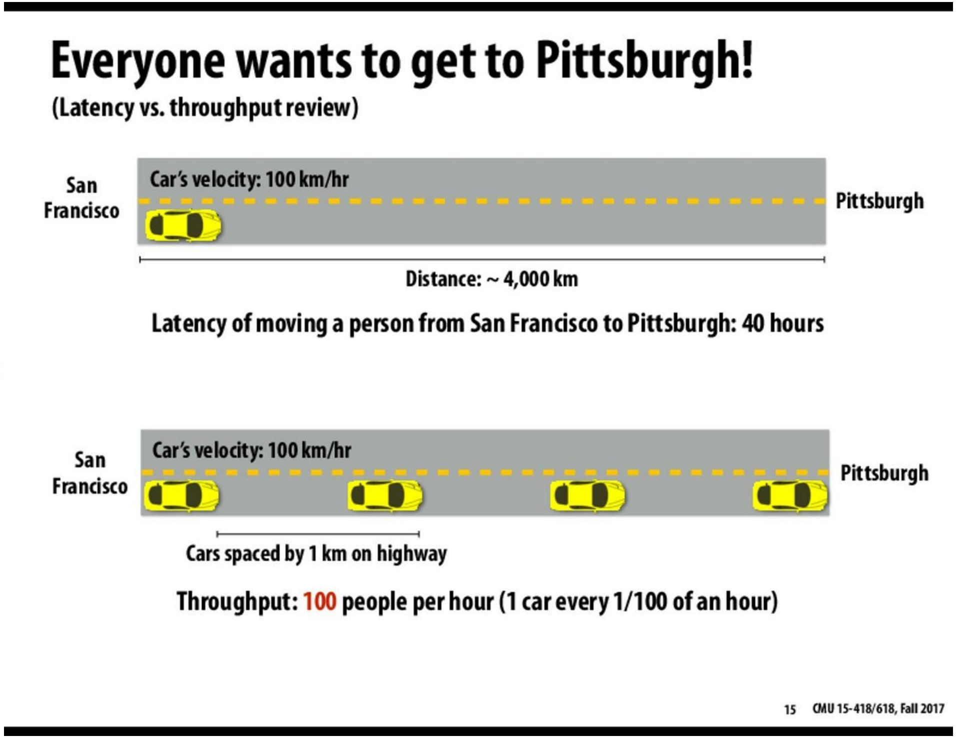
- Performance attributes in digital circuits
- Quantifying timing performance
- Static timing analysis, its differences with simulation, and why we care
- Optimizations: RTL-level ⟶ pipelining, parallelization and others
- Optimizations: Using primitives (e.g., DSP cores)

Performance Attributes in Digital Circuits

- The performance of a digital design is typically characterized in a few dimensions:
  - <u>Accuracy:</u> this is application-level work, the digital designer typically can't do much here, it's the job of the "algorithms engineer" to ensure that accuracy is within specs and the digital engineer simply translates that algorithm to a hardware implementation
    - (Part 1 of the course (now) covers the work of the digital designer, part 2 will cover the algorithms part)
  - <u>Timing:</u> throughput (how many outputs per second) and latency (worst delay from input to output)
  - Power, area, mechanical, thermal, safety, reliability, tampering, "rad-hard"ness, …
- Timing and accuracy are common attributes in all digital design projects. The rest are a bit advanced for this course, and may or may not be important depending on project specs.
- We'll focus on timing analyses and optimizations in this course.

Performance Attributes in Digital Circuits

- While throughput and latency might be tightly connected in some designs, they are actually separate design goals. The traffic example on the right is a great analogy →→→
- FPGAs (and digital circuits in general) are typically used for achieving extremely low latency levels compared to CPUs / GPUs
- Throughput is more a factor of input and output configurations



img src: https://medium.com/@nbosco/latency-vs-throughput-d7a4459b5cdb

Outline

- Performance attributes in digital circuits
- Quantifying timing performance
- Static timing analysis, its differences with simulation, and why we care
- Optimizations: RTL-level $\longrightarrow$ pipelining, parallelization and others
- Optimizations: Using primitives (e.g., DSP cores)

Outline

- Performance attributes in digital circuits
- Quantifying timing performance
- Static timing analysis, its differences with simulation, and why we care
- Optimizations: RTL-level ⟶ pipelining, parallelization and others
- Optimizations: Using primitives (e.g., DSP cores)

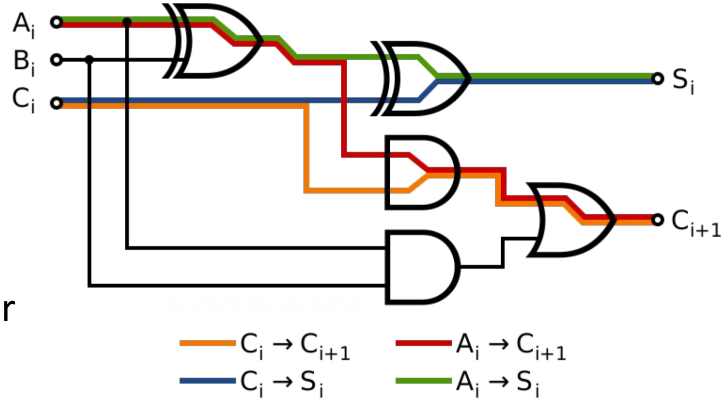
Quantifying Timing Performance

- How do we compute throughput and latency?

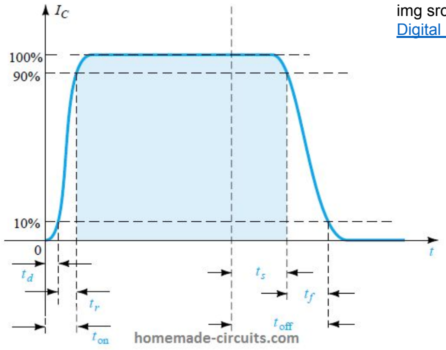
- Let's consider a simple combinational circuit →→

- Due to gate propagation delays, the correct response of the circuit output w.r.t. a change at the input appears after a non-zero time interval. Before that → all bets are off! The output can be "anything" (for the digital designer)

- *We* know what's happening here from our analog courses though: The signals are "slowly" rising or falling + there's noise, so the digital designer can't know whether a given signal is a 0 or a 1 before the signal "settles". That's why things are not predictable for the digital world.



img src: wikibooks, Digital Circuits



Quantifying Timing Performance

- Let's name this propagation delay based "finite waiting time" for the combinational circuit: t_p , this is equal to the "latency" in this simple circuit

- Therefore, for a purely combinational circuit, interpreting timing performance is simple: the input shouldn't be changed faster than $1/t_p$ and throughput = 1/latency here

- Things start getting more complicated when sequential components are added:

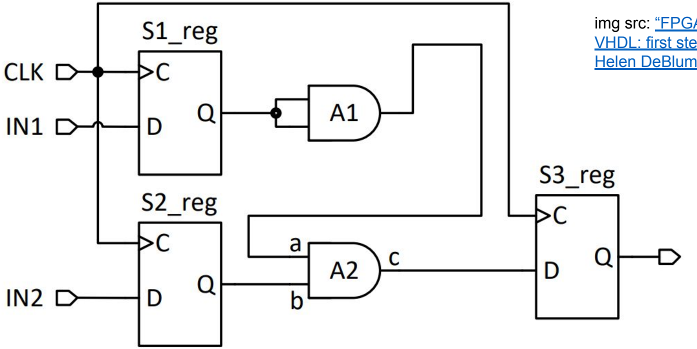
  - Signals inside the circuit now get **registered** at clock events rather than being available to read at arbitrary times (e.g., we connect the combinational circuit to a flip-flop and consider the output of the flip-flop as the useful signal instead)

  - We now have to analyze the timing performance of the register (clocked flip-flop), and throughput and latency get computed in terms of clock freq and periods, respectively

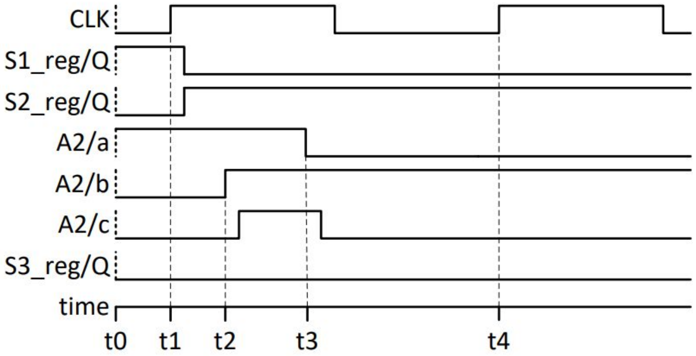
Quantifying Timing Performance

- Let's consider this simple combinational + sequential circuit with FFs at its IOs and a few gates in between →→→
- A simulation run for this circuit with realistic timing information demonstrates 3 important effects:

  1. FF propagation delay: S1_reg/Q and S2_reg/Q changing <u>after a short time</u> following the rising-edge of CLK
  2. Combinational propagation and net (wire) delays: A2/a takes a longer time to change compared to A2/b
  3. Solving glitches with FFs: A2/c should never have been high (from behavioral PoV), but the delays caused a glitch. The clock rising edge being at t4 solved this.

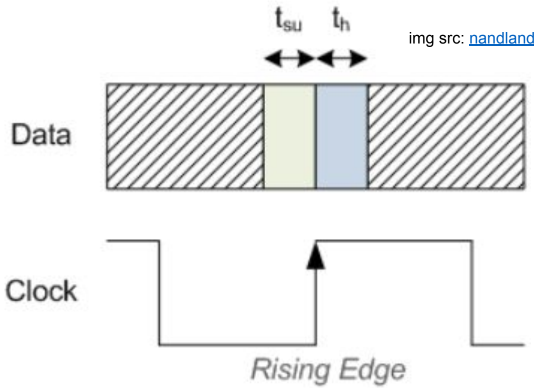


img src: "FPGAs with VHDL: first steps", Helen DeBlumont



Quantifying Timing Performance

- We knew about 2, but 1 and 3 are relatively new.
- To formalize our understanding of 1, we first need to define "setup" and "hold" times for FF timing
- "Setup time": The input to a flip-flop has to be stable for a certain amount of time before a clock event occurs
- "Hold time": The output of the flip-flop needs a certain amount of time before it settles (becomes stable)

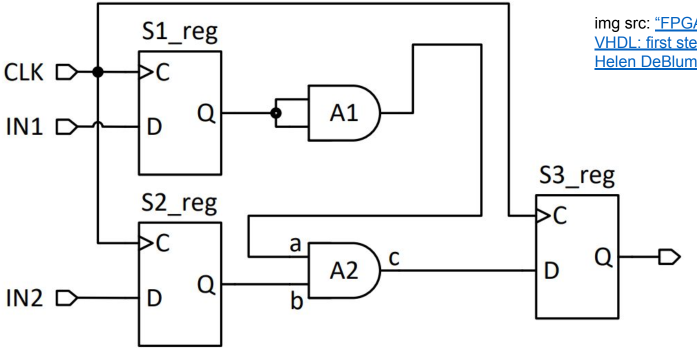


img src: nandland.com

- Note the similarity with the combinational circuit case here (the adder example)!!
  The FF is just getting some special definitions for the same thing: Since the clock rising edge is slow (doesn't happen instantly), we need time before and after it to talk about 0s and 1s.

Quantifying Timing Performance

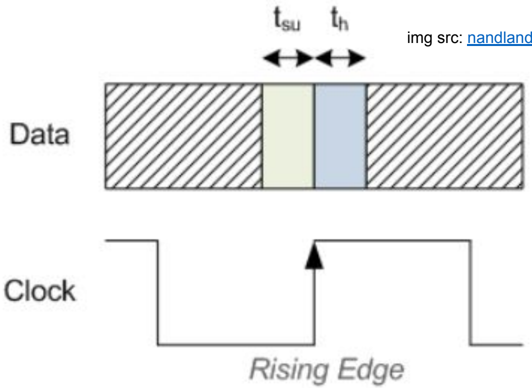
- OK, now we can formalize 3: considering setup and hold, looking at the timing diagram, we can now see how we can choose the max. clock speed



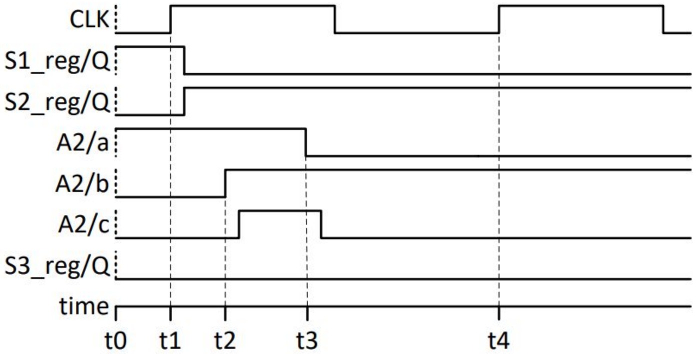
img src: "FPGAs with VHDL: first steps", Helen DeBlumont

- We determine the shortest clock interval looking at setup, hold and propagation times, specifically:

$$t_{clk\ (min)} = t_{su} + t_h + t_p$$



img src: nandland.com



- A faster clock risks various issues (next slide)

Quantifying Timing Performance

- Specifically, if we further increase clock frequency, we risk getting two things:
  - **Logic breakdown:** Real outputs simply don't match with behavioral simulation. This could happen if the second clock edge came before A2/c settled back to 0 in the previous slide.
    - Remember: signal changes seem like they happen instantly in behavioral sim since there's no timing info (we know this is wrong, but behavioral sim runs fast, so we use it to check our code)
  - **Or even worse, metastability:** if, e.g., the A2/c falling edge reaches the flip-flop input at a time instant that is very close to the clock rising edge, we risk falling into a state in which there's physically no way that we can know which value the flip-flop holds.
- Remember the lab2 prep lecture and e-mails: with any async input to a system (like a button press), we have the risk of metastability since we don't know when that input will rise/fall with regards to the clock edge. "Double-flopping" reduces the probability of this happening.

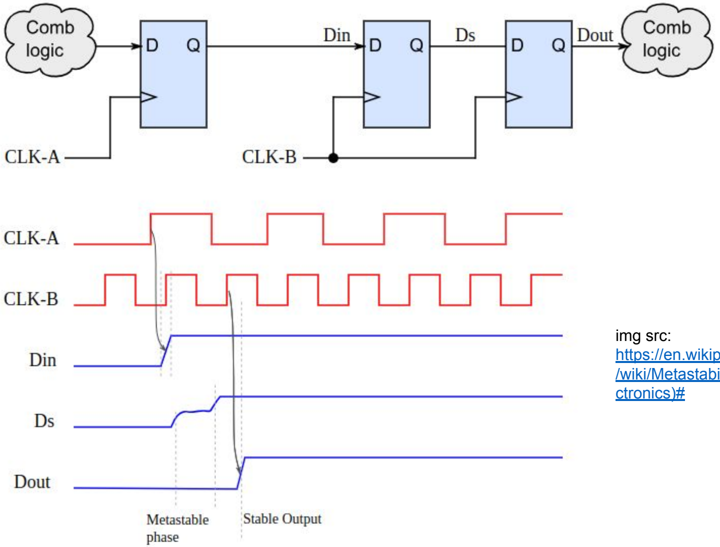
Quantifying Timing Performance

- Some more info on metastability:

  - It's caused by the finite transients of our signals (digital is just an abstraction here, we have analog signals "underneath")

  - There's always going to be a possibility of having metastability events, but we can lower that probability by 2/3/4/…-flopping

  - Can happen on clock domain crossing as well as async inputs like buttons, switches etc.



img src: https://en.wikipedia.org/wiki/Metastability_(electronics)#

It's not possible to simulate metastability in Vivado since we're already in the digital domain! Metastability is an analog phenomenon

Quantifying Timing Performance

- Back to quantification: Once we have the maximum clock frequency set, we know how much throughput our design can produce → if it's giving an output at every clock cycle, then throughput = clock frequency and max. latency = 1 clock period.

- There's a catch here though: Remember the tclk(min) equation → Adding large combinational circuits in between two clocked flip-flops forces us to slow the clock down for safe operation (lower throughput) since propagation and route delays are increased

- There are workarounds to this (i.e., parallelization, pipelining, …) which allow us to trade latency off for throughput. We'll cover these in the optimizations section.

- This situation is typical of timing analyses and optimizations → Setup and hold times are typically fixed for a given flip-flop in a given FPGA, so we try to minimize the propagation delays as well as the route delays to safely increase clock freq (hence, timing performance).

Outline

- Performance attributes in digital circuits
- Quantifying timing performance
- Static timing analysis, its differences with simulation, and why we care
- Optimizations: RTL-level $\longrightarrow$ pipelining, parallelization and others
- Optimizations: Using primitives (e.g., DSP cores)

Outline

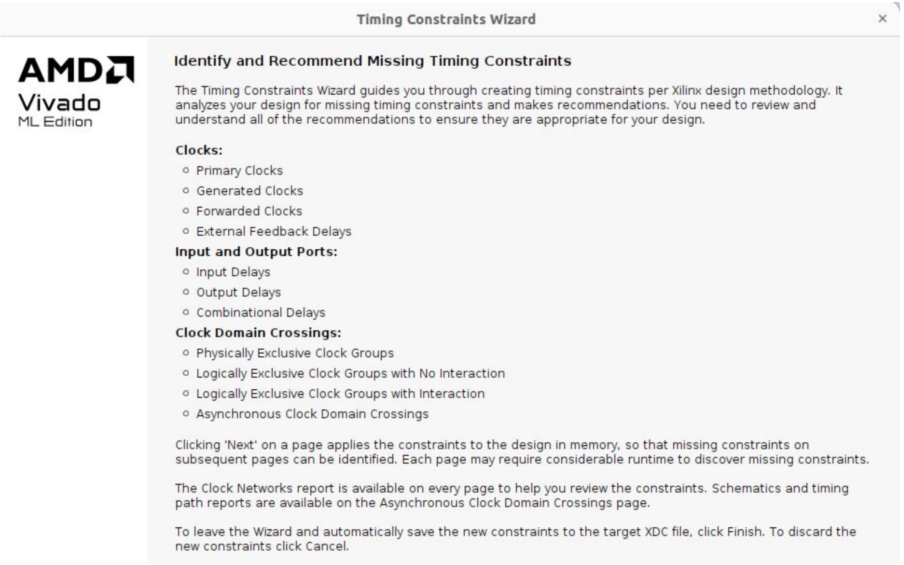
- Performance attributes in digital circuits
- Quantifying timing performance
- Static timing analysis, its differences with simulation, and why we care
- Optimizations: RTL-level ⟶ pipelining, parallelization and others
- Optimizations: Using primitives (e.g., DSP cores)

Static Timing Analysis

- OK we've now formalized timing performance with setup, hold and propagation delays.
- There's of course no way we can compute timing data manually for large designs, we'll have Vivado do this instead.
- This is called "static timing analysis": Static because there is no stimulus.
- In simulation (consider the post-synth and post-impl simulations, not behavioral) we generate certain stimuli to feed to the DUT, and we check the respective DUT output.
- While simulation is informative, if the stimulus doesn't trigger a worst case scenario (e.g., a late / early signal change on the critical path), then we'll simply miss that and not be able to fix it.
- Static timing analysis computes worst case delays on all paths analytically and checks this against timing constraints. **Lesson** → **we need both sim and STA.**

Static Timing Analysis

- "constraints" ?? .xcd files! →→→→→→
- We only specified IO pins (physical constraints) and the clock in the .xcd file up to now
- Vivado was having a field day up to now 🕺 💃 not doing any considerable optimizations and grabbing whatever resource it needs since we didn't constrain much.
- We will now try to tell it things like "we can't have that much delay between input A and output B!" and it will try re-running synthesis and implementation to account for those.



Static Timing Analysis

- Sometimes Vivado will be able to do some magic in synthesis and implementation to satisfy those constraints without changing circuit behavior (e.g., re-route wires, use a different arrangement of components), but sometimes it will just not be able to satisfy the timing constraints

- When this happens, we "fail timing":

- This means, through some statistical calculations that Vivado did based on a set of process / voltage / temperature (PVT) assumptions, some signals did not reach their destinations (i.e., from input A to output B) in the time allocated by the constraints **we enforced**.

- The amount of time remaining for a signal is called "slack", and if slack is negative, timing fails. There are many types of slack Vivado computes (WNS, TNS, THS, WHS, WSPS, TSPS, …)

Static Timing Analysis

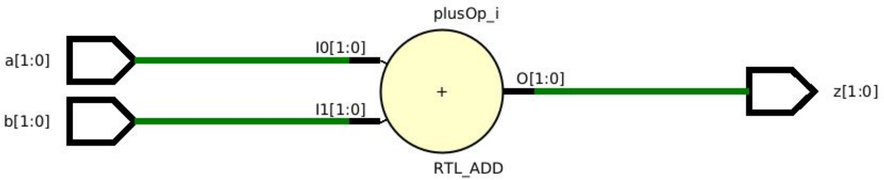
- Let’s consider a super-simple design and see how constraints work

- VHDL (DUT):

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.std_logic_unsigned.all;

entity ttt is
    Port (
        a, b : in  STD_LOGIC_VECTOR(1 downto 0);  -- 2 inputs 2-bit
        z    : out  STD_LOGIC_VECTOR(1 downto 0) -- 1 output 2-bit
        );
    end ttt;

architecture Behavioral of ttt is
begin
    z <= a + b;
end Behavioral;
```

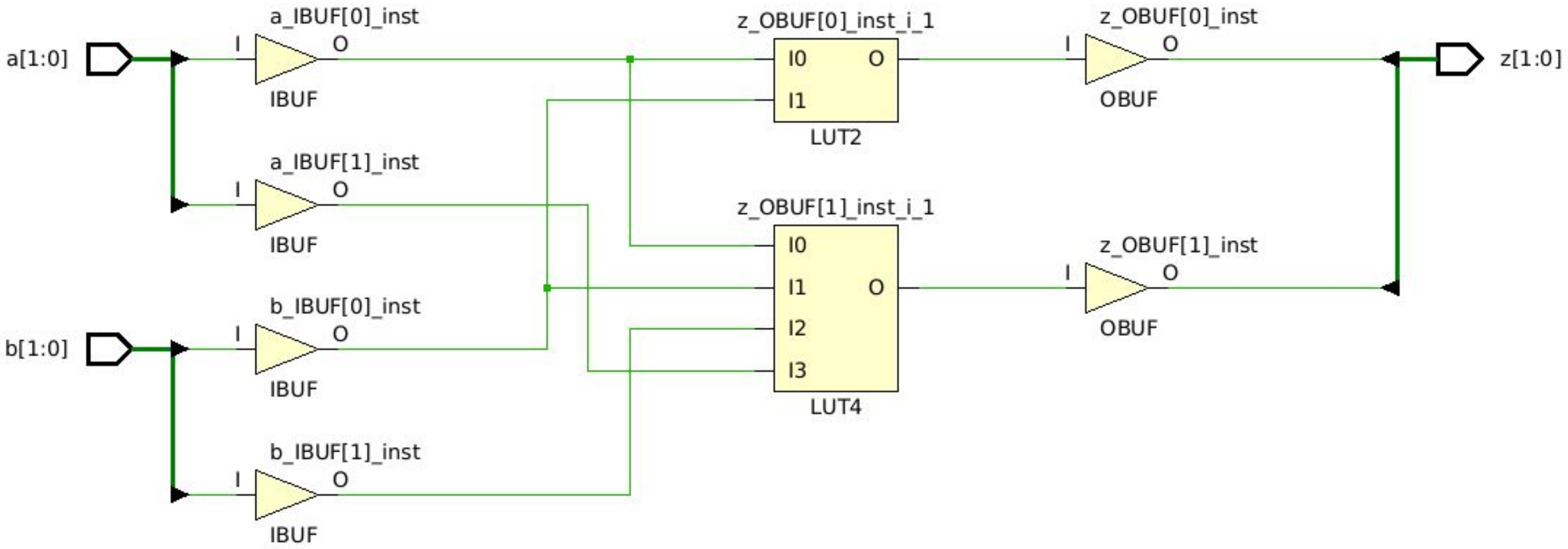
- .xdc (without timing constraints):

```
set_property -dict {PACKAGE_PIN V17 IOSTANDARD LVCMOS33} [get_ports {a[0]}]
set_property -dict {PACKAGE_PIN V16 IOSTANDARD LVCMOS33} [get_ports {a[1]}]
set_property -dict {PACKAGE_PIN W16 IOSTANDARD LVCMOS33} [get_ports {b[0]}]
set_property -dict {PACKAGE_PIN W17 IOSTANDARD LVCMOS33} [get_ports {b[1]}]

## LEDs
set_property -dict {PACKAGE_PIN U16 IOSTANDARD LVCMOS33} [get_ports {z[0]}]
set_property -dict {PACKAGE_PIN E19 IOSTANDARD LVCMOS33} [get_ports {z[1]}]

## Configuration options, can be used for all designs
set_property CONFIG_VOLTAGE 3.3 [current_design]
set_property CFGBVS VCCO [current_design]

## SPI configuration mode options for QSPI boot, can be used for all designs
set_property BITSTREAM.GENERAL.COMPRESS TRUE [current_design]
set_property BITSTREAM.CONFIG.CONFIGRATE 33 [current_design]
set_property CONFIG_MODE SPIx4 [current_design]
```



Static Timing Analysis

- This synthesizes to a simple no-carry 2-bit adder as expected (LUTs realize adder truth tables):



Static Timing Analysis

- Say we add a 1 ns timing constraint on the path from input a[0] to output z[0] (unrealistic):

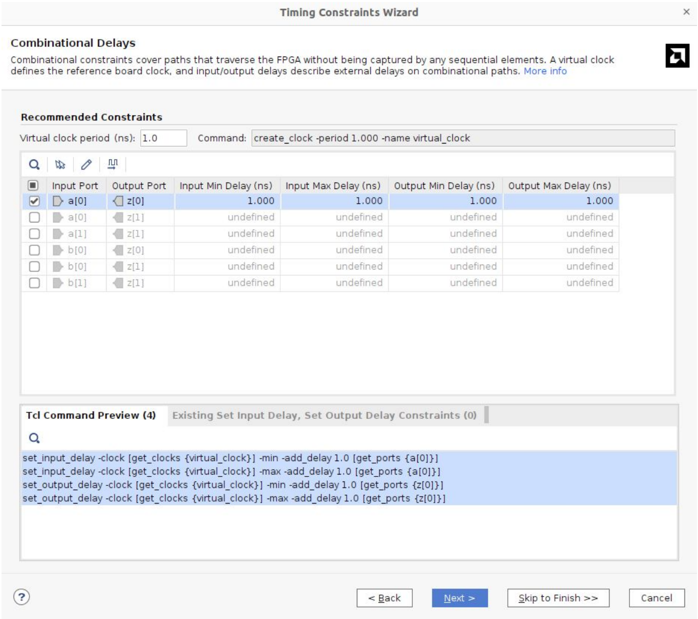
```
create_clock -period 1.000 -name virtual_clock
set_input_delay -clock [get_clocks virtual_clock] -max -add_delay 1.000 [get_ports {a[0]}]
set_output_delay -clock [get_clocks virtual_clock] -max -add_delay 1.000 [get_ports {z[0]}]

```

- We can do this via the “Constraints Wizard” →→→

- Since timing violations get detected over clock edges in Vivado, the wizard creates a “virtual clock” at 1 GHz freq, and then times the combinational path accordingly

- After adding this constraint and re-running implementation, the timing report summary shows that we failed this constraint with negative slack.

WNS	TNS
-8.808	-8.80



Static Timing Analysis

- This constraint was just added to show us what a timing failure looks like though. In most designs we will not be setting such individual timing constraints on pathways in the circuit.

- We will rather be concerned with "how fast we can clock" that circuit, input and output delays, and the worst latencies from the inputs to the outputs.

- In most cases, a timing failure will mean we have too much combinational logic between two given flip-flops, and we will need to find solutions to that problem.

- To make it easier to characterize such problems, keeping designs hierarchical, i.e., breaking the circuit into numerous entities, and analyzing these entities individually helps a lot.

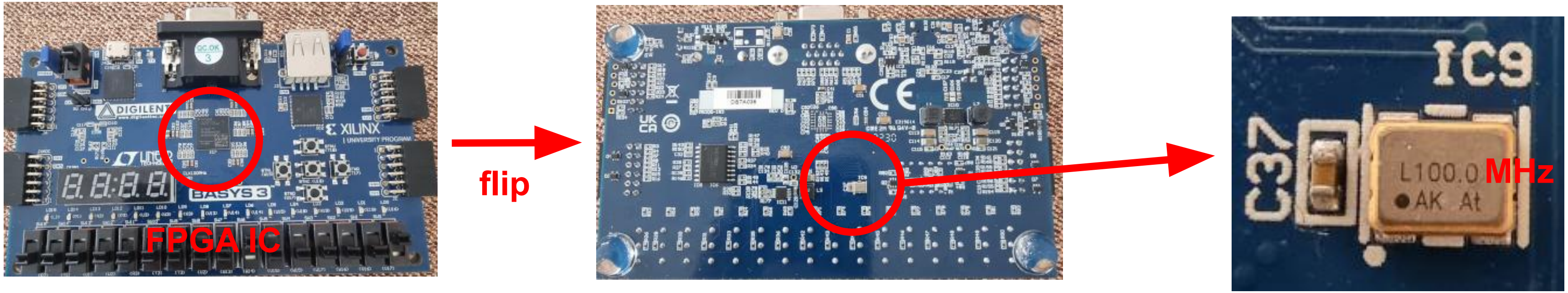
- This is what we did with the debouncer (separate entity)! We didn't analyze anything there, but that was a good example of hierarchical design.

Static Timing Analysis

- So how do we add clock, input delay and output delay constraints? We know how to add the clock constraint already! Recall these lines from earlier .xcd files:

```
## Clock signal
set_property -dict { PACKAGE_PIN W5   IOSTANDARD LVCMOS33 } [get_ports clk]
create_clock -add -name sys_clk_pin -period 10.00 -waveform {0 5} [get_ports clk]
```

- Line 1 says "the FPGA will be receiving a signal named clk on pin W5", this is a physical constraint like with those switches and LEDs earlier (tells the FPGA to expect the clock signal at that pin). W5 is connected to the oscillator on the Basys3 board (outside the FPGA IC, but still on the board):



Static Timing Analysis

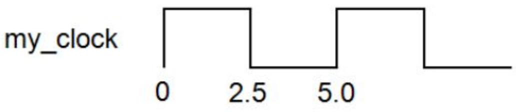
- Line 2 is the timing constraint

```
## Clock signal
set_property -dict { PACKAGE_PIN W5   IOSTANDARD LVCMOS33 } [get_ports clk]
create_clock -add -name sys_clk_pin -period 10.00 -waveform {0 5} [get_ports clk]
```

- From AMD:

  - Clocks are created with the create_clock Tcl command
    - create_clock -name <name> -period <period> <objects>
    - <period> is the period of the clock
    - <name> is the user assigned name for the clock
    - <objects> are the list of pins, ports, or nets to which to attach the clock

```
create_clock -name my_clock -period 5.0
```




```
create_clock -name clk2 -period 5.0 -waveform {1.0 4.0}
```

img src: https://xilinx.github.io/xup_fpga_vivado_flow/presentations.html

Static Timing Analysis

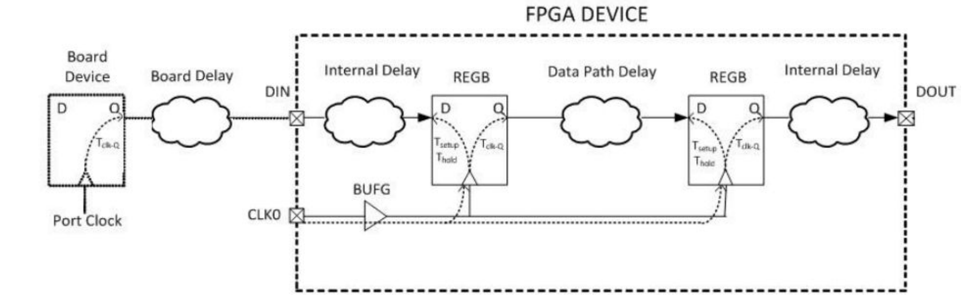
- On top of the clock, there are input delays and output delays that must be considered.
  - Input delay: due to travel of input signals from input devices to the FPGA
  - Output delay: due to travel of outputs from the FPGA to the "external device" that uses the outputs.

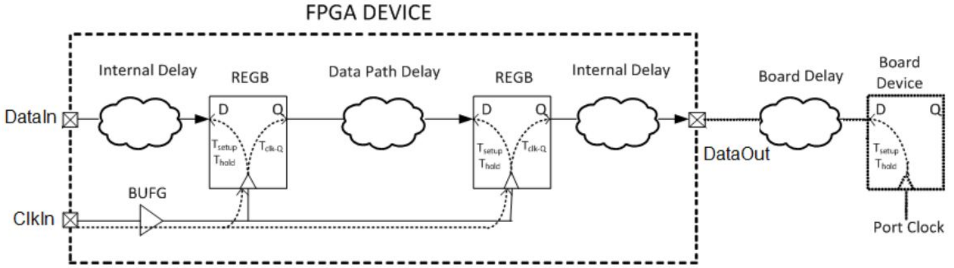
- We may have a hard time estimating good numbers for these, but be aware that Vivado still needs them to be able to give a good estimate about the real scenario! (default =0)

- An example to clarify the need for IO delays (async reset):
  - Imagine a power electronics control circuit running with a fast 100 MHz clock
  - The circuit needs to be reset within 2 clock cycles (20 ns) if an emergency happens at the actuator
  - The circuit gets notified of a reset with an async pulse coming from a separate detector circuit 2 m away, connected by coax cable (incurring approx. 8.3 ns of propagation delay)
  - Worst propagation delay inside the FPGA from the reset pin to registers that provide output is 13 ms
  - If this coax input delay was not considered during sim (i.e., reset triggered at t=0, not t=8.3ns), simulations will pass but the actual test will fail with a 13+8.1 = 21.1 ns delay for a reset since it's over 20 ns.

Static Timing Analysis

- With clock, input delay and output delay constraints, Vivado is ready to run STA and tell us whether timing fails or not
- Since Vivado analyzes timing from FF to FF, clock constraints are straightforward
- For analyzing IO timing constraints Vivado assigns fake FFs at the input and output like it did in the analysis of the purely combinational circuit (recall the "virtual_clock")





img src: https://xilinx.github.io/xup_fpga_vivado_flow/presentations.html

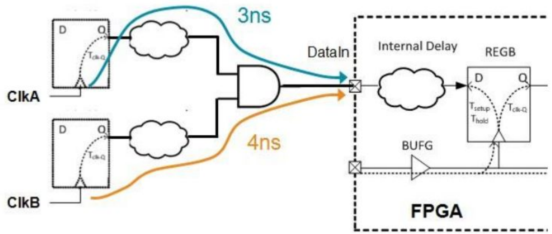
Static Timing Analysis

- Just like create_clock, input and output delays are set with a command on the .xcd file
- Commands to set input delays:

- `set_input_delay -clock <clock_name> <delay> <objects>`
  - `<clock_name>` is the name of the clock used by the external device
    - Can be a real or virtual clock
    - Can be the ***name*** of a clock; does not need to be a clock object
      - Can use a clock object if desired
  - `<objects>` is the list of objects to which to attach the `set_input_delay`
    - Usually a set of input and/or inout ports
    - Usually uses the `get_ports` command or the `all_inputs` command
  - `<delay>` is the delay from `<clock_name>` to the attached `<objects>`
    - Includes the external device and board delay

- An input can have multiple `set_input_delay` commands associated with it
  - Use the `-add_delay` option
  - Results in multiple static timing paths to check

```
set_input_delay -clock ClkA 3 [get_ports DataIn]
set_input_delay -clock ClkB 4 [get_ports DataIn] -add_delay
```

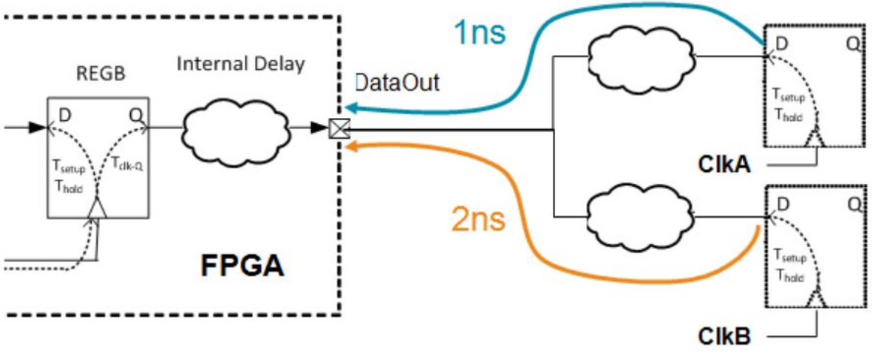


img src: https://xilinx.github.io/xup_fpga_vivado_flow/presentations.html

Static Timing Analysis

- Output delays are similar

```
set_output_delay -clock ClkA 1 [get_ports DataOut]
set_output_delay -clock ClkB 2 [get_ports DataOut] -add_delay
```



img src: https://xilinx.github.io/xup_fpga_vivado_flow/presentations.html

- `set_output_delay -clock <clock_name> <delay>`
  - `<clock_name>` is the name of the clock used by the external device
    - Can be a real or virtual clock
    - Can be the *name* of a clock; does not need to be a clock object
      - Can use a clock object if desired
  - `<objects>` is the list of objects to which to attach the `set_output_delay`
    - Usually a set of output and/or inout ports
    - Usually uses the `get_ports` command or the `all_outputs` command
  - `<delay>` is the delay from the attached `<objects>` to the external device's clock
    - Includes the external device's requirements and board delay

Static Timing Analysis

- Recap:
  - Set clock constraints
  - Set IO delay constraints

    (the constraints wizard helps us out, but we can use commands directly too)
  - (re-)Run synthesis / implementation (they both run STA) to get updated results
  - Check timing reports to see if the timing fails
  - Check which paths failed on the report and proceed to optimization

Outline

- Performance attributes in digital circuits
- Quantifying timing performance
- Static timing analysis, its differences with simulation, and why we care
- Optimizations: RTL-level $\longrightarrow$ pipelining, parallelization and others
- Optimizations: Using primitives (e.g., DSP cores)

Outline

- Performance attributes in digital circuits
- Quantifying timing performance
- Static timing analysis, its differences with simulation, and why we care
- Optimizations: RTL-level ⟶ pipelining, parallelization and others
- Optimizations: Using primitives (e.g., DSP cores)

Optimizations

- STA is not enough on its own, it only locates the timing problems in the circuit. We need a solution to those problems in the form of concrete optimizations.

- The topic of timing optimization is ***vast***, with all sorts of heuristics and crazy tricks

> large FPGAs with multiple SLRs. Closing timing is always a head-ache. One of our designs actually has a bit of logic in it to track the relative phases of two clocks to decide whether to sample on the rising edge or the falling edge for any given data beat.

pencan • 5mo ago • Edited 5mo ago

> One common technique is leveraging useful skew, meaning to delay the clock so as to give a longer setup time. https://en.wikipedia.org/wiki/Clock_skew.

F_P_G_A • 2y ago

> With any of the RF/filtering/processing type IP blocks, one way to increase optimization is to run the IP at a multiple of the sample rate. For example, I frequently run FIR filters at 3X or 4X the sample rate. If the sample rate is 50 MSPS, try running it at 200 MHz. The Xilinx tools are smart enough to figure out the resource sharing. The

Optimizations

- In this part of the course (part 1), we will cover optimizations which make absolutely no changes to the behavioral characteristics (input-to-output logic) of our circuit.
- In part 2 of the course, we will focus on methods that make such changes (e.g., navigating the accuracy vs. timing trade-off, using less bits for the same calculation by sacrificing some accuracy etc.)
- The ones we will cover now are the most common ones (not exhaustive of course):

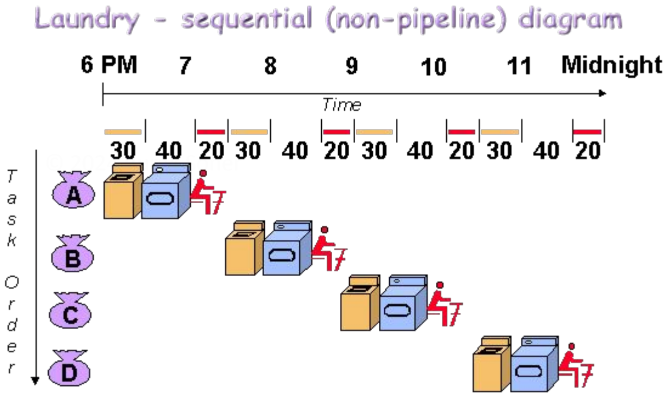
1. RTL-level optimizations: pipelining and parallelism
2. Vivado optimization tricks (some settings and strategies for synth. / impl.)
   (this one is a bit advanced and it gives diminishing returns, we won't dwell on it too much)
3. Using design primitives to replace inefficient parts of bare RTL

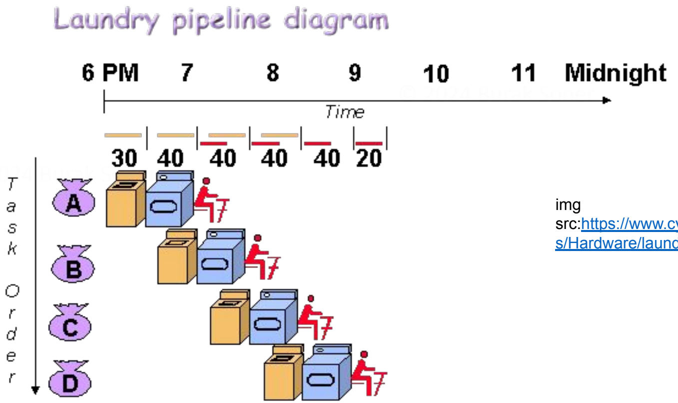
Optimizations - 1) RTL-Level

- Before we start with RTL-level optimizations, there's one thing that we need to recall. When we are writing RTL code…
  - we are not wiring FPGA primitives to realize our circuit (i.e., we are not doing implementation)
  - we are not drawing a netlist for our circuit (i.e., we are not doing synthesis)
  - we are not even giving a complete description of our circuit elements!
- We are simply writing a **behavioral description** of our circuit. Vivado reads that description and **interprets** it (via synth. + impl.) to do the above. So we only **advise** Vivado with our RTL.
- However, since Vivado is not a perfect optimizer, the more specific we get about how our circuit should be, the closer we'll get Vivado to generate that implementation.
- Today, Vivado is pretty good at "**inferring**" certain optimizations itself, but it might still need our help. Make sure you're telling it the right thing, because Vivado won't double-check!

Optimizations - 1) RTL-Level

- For each method we'll make a brief definition and go through an example.
- Pipelining → there's this well-known laundry analogy:





img src:https://www.cybercomputing.co.uk/Languages/Hardware/laundryAnalogy.html

- Replace washing, drying, folding with some operations on the FPGA, replace their processing times with propagation delays, and that's exactly what we'll be doing on the FPGA

Optimizations - 1) RTL-Level

- Note how we need exclusive operations for this, i.e., if the person who does the folding were washing the clothes by hand, we wouldn't be able to do this optimization.

- A hint from the laundry example: *"…notice that although the washer finishes in half an hour, the dryer takes an extra ten minutes, and so the wet clothes must wait ten minutes for the dryer to free up."*

  - this implies that we need additional memory to temporarily store the outputs of pipelined operations before they are used in the next stage (FFs in between stages)

- Another hint from the laundry example: *"…the* ***length of the pipeline is dependent on the length of the longest step.*** *It is therefore most efficient to have small equally sized steps in processing so that efficient pipelining can be incorporated…"*

  - "length of the pipeline" refers to latency here

Optimizations - 1) RTL-Level

- **Pipelining example**: cascaded multiplication
- Single stage (no pipelining):

```
--process for calcultation of the equation.
PROCESS(Clk,a,b,c,data)
BEGIN
    if(rising_edge(Clk)) then
    --multiplication is done in a single stage.
        result <= a*b*c*data;
    end if;
END PROCESS;
```

Why would we pipeline this? Because those 3 mults in a single rising edge mean a lot of logic between two FFs, which means smaller max. clock speed without any timing or logic violations (recall slide 15)

Img src: https://vhdlguru.blogspot.com/2011/01/what-is-pipelining-explanation-with.html

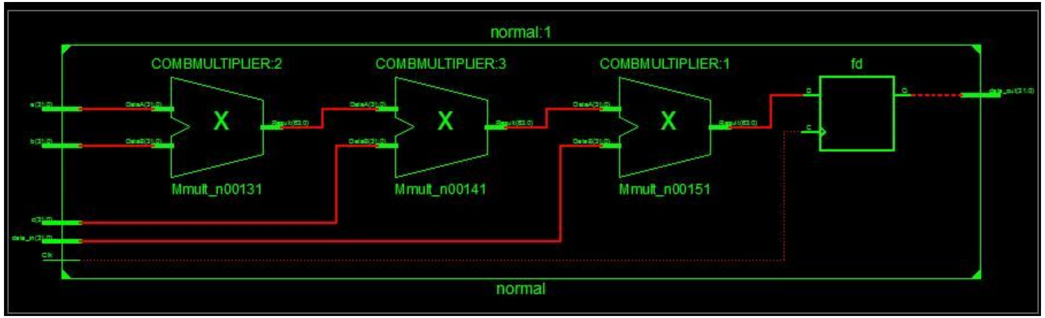
- To trigger a pipelined implementation, we explicitly define new temporary variables

```
--process for calcultation of the equation.
PROCESS(Clk)
BEGIN
    if(rising_edge(Clk)) then
    --Implement the pipeline stages using a for loop and case statement.
    --'i' is the stage number here.
    --The multiplication is done in 3 stages here.
    --See the output waveform of both the modules and compare them.
        for i in 0 to 2 loop
            case i is
                when 0 => temp1 <= a*data;
                when 1 => temp2 <= temp1*b;
                when 2 => result <= temp2*c;
                when others => null;
            end case;
        end loop;
    end if;
END PROCESS;
```

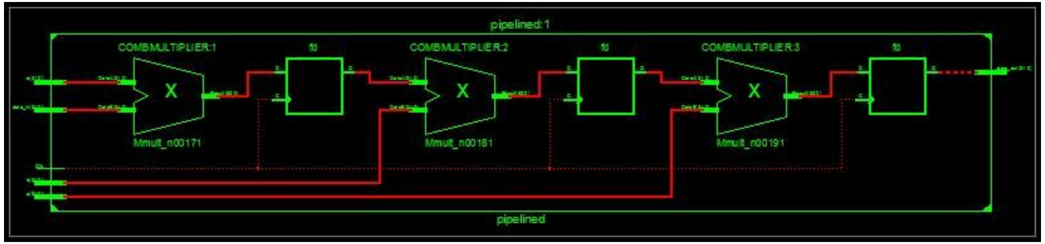
Recall: this is VHDL, it’s not software, so don’t think of this loop as running line-by-line. This is just saying: “synthesize 3 serially connected multiplications, each triggered by a clock rising edge”, we could unroll this for loop and write each line separately too, it would have the same effect

Optimizations - 1) RTL-Level

- These synthesize to the following (squares: FF, mux-like blocks: mults)



single-stage (not pipelined)



3-stage (pipelined)

Img src: https://vhdlguru.blogspot.com/2011/01/what-is-pipelining-explanation-with.html

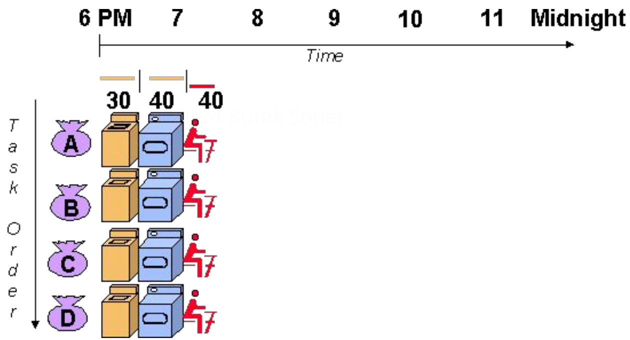
Optimizations - 1) RTL-Level

- So what's the result? What did we gain by doing this? The datapath looks pretty much the same with 3 mults in cascade, just new FFs came in…
- The pipelined implementation has a lot less combinatorial logic in between flip-flops, so we can clock the whole thing at much larger clock frequencies (without timing or logic violations)
- However, we increased the number of clock cycles that it takes for a given input to pass through the system (because FFs get clocked sequentially, one after the other) by 3x, this means an increase in latency in terms of the number of clock cycles.
- But don't miss the catch here! We have pushed the max. clock frequency higher, so the latency might have even dropped if the clock freq could be increased by more than 3x (we would need to run STA on this design to find out if this holds).
- Extra: There's some implication that Vivado might infer pipelining based on constraints so that the designer doesn't have to explicitly re-design the RTL for pipelining, but I personally feel like that's not possible with the way Vivado currently handles optimization. This seems to be an open issue.

Optimizations - 1) RTL-Level

- Pipelining is **one** form of parallelization. Specifically, it's a form where we don't add extra resources (e.g., an additional washer+dryer+folder) to the system, but we cleverly utilize the idle times of existing resources to increase throughput.

- Other forms of parallelization are possible, for instance if we had 4 washers, 4 dryers and 4 folders, we would finish the whole thing in 1 cycle! →→



- This is called an "embarrassingly parallel" problem, we can just throw more resources in to solve it (nothing inherently embarrassing about it though, just poor terminology)

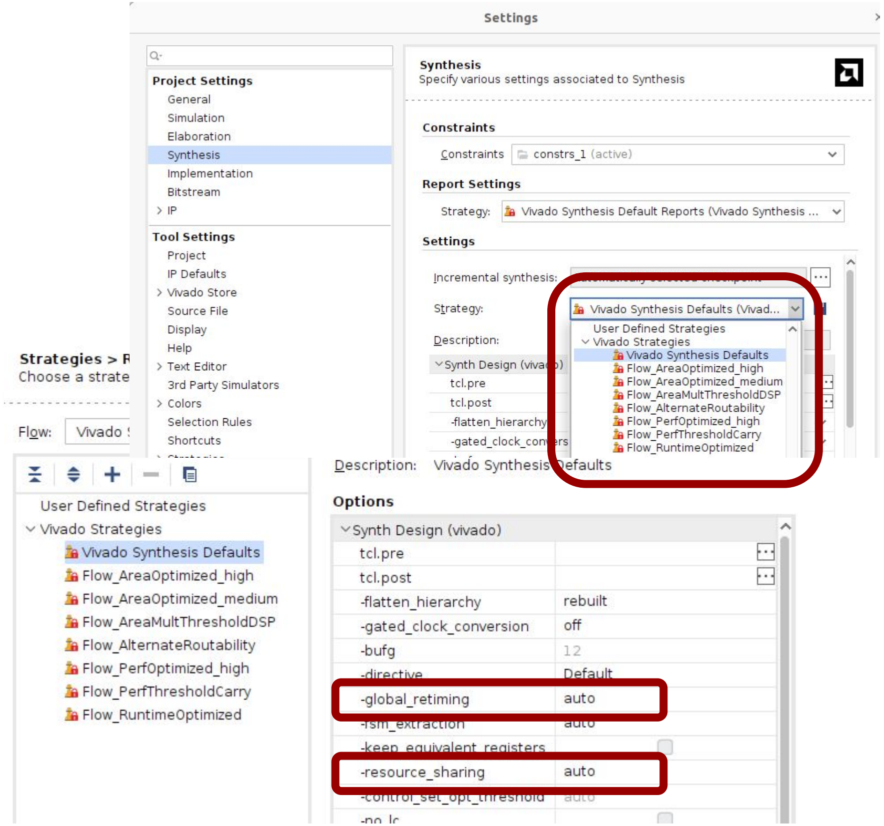
- In our domain, these are called SIMD (single instruction multiple data) problems. Canonical example is image processing (you apply the same operations on every pixel). This family of problems is what popularized GPUs when they first came out, and they're naturally amenable to FPGAs.

Optimizations - 1) RTL-Level

- **Parallelization example**: well this is very straightforward, just create another process !
- VHDL is called a "parallel language" for this. Processes synthesize to circuits that run (exist?) in parallel. However, we need problems that are behaviorally parallelizable for this to work.
- For instance, in the cascaded mult problem, we can do a*b in one process, c*data in another, in parallel, but that's about it. We will still need the 3rd mult to get (a*b)*(c*data).
- Even with this small trick we've gained 1 clock cycle (3 was needed earlier, now it's 2 since the first mults are in parallel), but we haven't gained 3x, which was what embarrassing parallelization promised
- Algorithmic conversion of such parallelizable problems is a research topic on its own, e.g., what I just described is a reduction algorithm. The parallelization we discuss here is simply the implementation of those conversions in hardware.

Optimizations - 2) Vivado Tricks

- Pipelining and parallelization are typically used to improve timing. However, going down this path, we might end up with using just too many resources on the chip, and maxing out FPGA capacity (area).
- If that is the case, resource sharing, some Vivado synthesis / implementation strategies and retiming might help us out.
- We will not go into details here since these are a bit advanced, but you can try them out by simply selecting them in the settings view before you run synth / impl, and checking utilization & timing reports after you do.



Outline

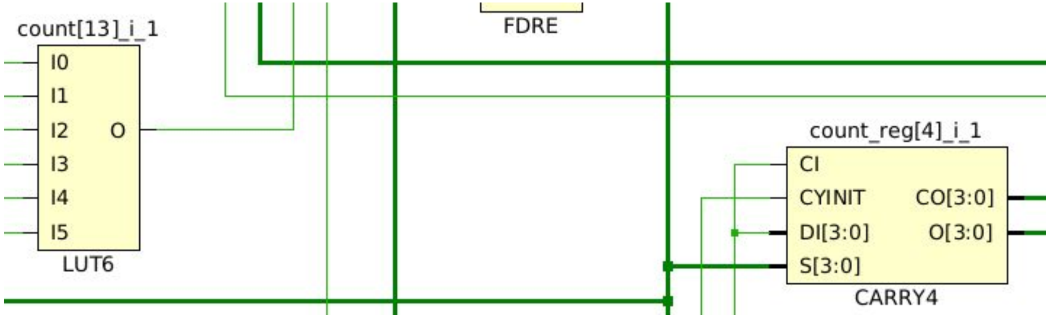
- Performance attributes in digital circuits
- Quantifying timing performance
- Static timing analysis, its differences with simulation, and why we care
- Optimizations: RTL-level → pipelining, parallelization and others
- Optimizations: Using primitives (e.g., DSP cores)

Outline

- Performance attributes in digital circuits
- Quantifying timing performance
- Static timing analysis, its differences with simulation, and why we care
- Optimizations: RTL-level ⟶ pipelining, parallelization and others
- Optimizations: Using primitives (e.g., DSP cores)

Optimizations - 3) FPGA Primitives

- The 3rd optimization is arguably the most important one: using primitives instead of bare RTL
- The FPGA is not *exactly* a bunch of gates and interconnects between them. It's much more heterogeneous, i.e., it has different types of specialized components inside called primitives
- We've seen this a few times by now: our adder circuits were synthesized into something called a CARRY block, and our custom combinational logic got synthesized into look-up tables of different input sizes, carrying the truth table of that logic block.



Optimizations - 3) FPGA Primitives

- However, those were the simplest primitives. There are many other complicated components, much like microcontroller peripherals (ADCs, transceivers, DSP blocks, …)

Advanced

Design Element	
XADC	Primitive: Dual 12-Bit 1MSPS Analog-to-Digital Converter

Arithmetic Functions

Design Element	
DSP48E1	Primitive: 48-bit Multi-Functional Arithmetic Block

Slice/CLB Primitives

Design Element	Description
CARRY4	Primitive: Fast Carry Logic with Look Ahead
CFGLUT5	Primitive: 5-input Dynamically Reconfigurable Look-Up Table (LUT)
LUT1	Primitive: 1-Bit Look-Up Table with General Output
LUT2	Primitive: 2-Bit Look-Up Table with General Output
LUT3	Primitive: 3-Bit Look-Up Table with General Output
LUT4	Primitive: 4-Bit Look-Up-Table with General Output
LUT5	Primitive: 5-Input Lookup Table with General Output
LUT6	Primitive: 6-Input Lookup Table with General Output
LUT6_2	Primitive: Six-input, 2-output, Look-Up Table
MUXF7	Primitive: 2-to-1 Look-Up Table Multiplexer with General Output
MUXF8	Primitive: 2-to-1 Look-Up Table Multiplexer with General Output
SRL16E	Primitive: 16-Bit Shift Register Look-Up Table (LUT) with Clock Enable
SRLC32E	Primitive: 32 Clock Cycle, Variable Length Shift Register Look-Up Table (LUT) with Clock Enable

Optimizations - 3) FPGA Primitives

- We can think of these primitives as the old-school gate-level ICs that we discussed in the first courses, the ones that people used to digital design with back in the 60s and onwards.
- The designer of the past has been replaced here by Vivado!
- We (high-level architect) give the designer a description (VHDL), and the designer figures out which pieces to put together and how, in order to optimally realize our description.
- Vivado can "**infer**" the most basic primitives reliably, and it can sometimes do it for the more complicated ones too, just by looking at our bare VHDL, without us explicitly calling that primitive out in our code.
- However, just like we saw earlier, Vivado is not the best designer ever (yet). Therefore sometimes we need to be more explicit in our descriptions as to how and where these primitives should be used

Optimizations - 3) FPGA Primitives

- **Primitives example:** the DSP48E1 primitive that does wide-bitwidth arithmetic can be inferred from simple RTL arithmetic in our VHDL code when the "`attribute use_dsp: string;`" directive is included in the architecture declaration, [but this doesn't always work](#).

- Vivado gives us an alternative: use "language templates" to embed the primitive into your RTL explicitly, and thus have full control over its behavior in your circuit. This is just like the debouncer primitive! The only difference is we had RTL for that one, but this DSP core is directly embedded in hardware (like an ASIC inside the FPGA)

```
DSP48E1_inst : DSP48E1
generic map (
   -- Feature Control Attributes: Data P
   A_INPUT => "DIRECT",                  --
   B_INPUT => "DIRECT",                  --
   USE_DPORT => FALSE,                   --
   USE_MULT => "MULTIPLY",               --
   USE_SIMD => "ONE48",                  --
   -- Pattern Detector Attributes: Patte
   AUTORESET_PATDET => "NO_RESET",       --
   MASK => X"3fffffffffff",              --
   PATTERN => X"000000000000",           --
   SEL_MASK => "MASK",                   --
   SEL_PATTERN => "PATTERN",             --
   USE_PATTERN_DETECT => "NO_PATDET",    --
   -- Register Control Attributes: Pipel
   ACASCREG => 1,                        --
```

direct instantiation, gives full control but it's a bit hard (parameter list goes on for 80 more lines!)

```
ADDMACC_MACRO_inst : ADDMACC_MACRO
generic map (
   DEVICE => "7SERIES",     -- Target Device: "7SERIES",
   LATENCY => 4,            -- Desired clock cycle laten
   WIDTH_PREADD => 25,      -- Pre-Adder input bus width
   WIDTH_MULTIPLIER => 18,  -- Multiplier input bus wi
   WIDTH_PRODUCT => 48)     -- MACC output width, 1-48
port map (
   PRODUCT => PRODUCT,        -- MACC result output, wid
   MULTIPLIER => MULTIPLIER,  -- Multiplier data input
   PREADDER1 => PREADDER1,    -- Preadder data input,
   PREADDER2 => PREADDER2,    -- Preadder data input,
   CARRYIN => CARRYIN, -- 1-bit carry-in input
   CE => CE,      -- 1-bit input clock enable
   CLK => CLK,    -- 1-bit clock input
   LOAD => LOAD, -- 1-bit accumulator load input
   LOAD_DATA => LOAD_DATA, -- Accumulator load data i
   RST => RST    -- 1-bit input active high synchrono
);
```

Vivado also provides pre-configured macros of these for certain operations, for instance this one is configured as a multiply accumulate block

Optimizations - 3) FPGA Primitives

- Using such components for optimization is a huge part of digital design work nowadays

- To be frank, this is thanks to the success of the FPGA providers like Xilinx (AMD) / Altera (Intel) who have set up their hardware such that their whole product line (even the earlier ones!!) is internally compatible because almost all of their FPGAs use the same hardware blocks (at least in terms of input-output configurations).

- So if someone built a very good VHDL library for, say, a UART module back in 2004, it's most probably still valid for FPGAs today, so you can just take the library and click synthesize.

- Examples of such ready-made components are FPGA primitives like these, pre-packaged IP cores (there are companies who design and sell just these "IPs", you can even encrypt them for licensing before giving it out etc.), or other RTL modules like our debouncer

- You will almost invariably use at least a few such modules in your term projects too.

next:

fixed-point arithmetic, pipelining optimizations

Part 2 of the course (applications, i.e., more exciting stuff than blinking LEDs)

🙋🏽‍♂️