ELEC 305

Digital System Design Lab

Fall 2024

Lecture 5:

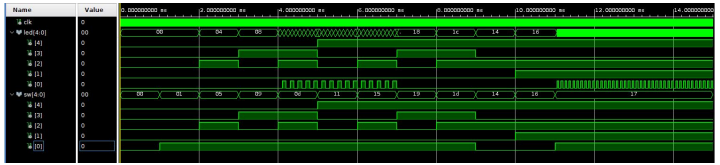
Number Formats and Arithmetics

Recap of Part 1

- 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
  - **simulate** DUT **behavior** in VHDL, characterize its accuracy, **constrain** and **analyze** its **timing** characteristics, and consequently **optimize** it if it fails
- This pretty much sums up the tasks of a digital designer, but there's a catch, a recurring theme → Vivado keeps getting better at doing most of these automatically. We therefore need to climb higher up on the abstraction ladder.
- The designer should also be proficient in building / testing / optimizing on the **algorithms** level to advise Vivado properly.
- That's what we'll be covering in Part 2.

```
1.  library IEEE;
2.  use IEEE.STD_LOGIC_1164.ALL;
3.
4.  entity  coffeemaker is
5.    Port ( clk : in  STD_LOGIC;
6.           led : out STD_LOGIC;
7.           sw  : in STD_LOGIC
8.         );
9.  end coffeemaker;
10.
11. architecture Behavioral of coffeemaker is
12.   signal pulse : std_logic := '0';
13.   signal count : integer range 0 to 199999999 := 0;
14. begin
15.   process(clk, sw)
16.   begin
17.        ...
18.   end process;
19.
20.   led <= pulse;
21. end Behavioral;
```

Failed Timing! -8.808 -8.80



Intro to Part 2

- The family of circuits we've dealt with so far are typically called "boolean circuits": Inputs and outputs were binary, and even when we used vectors we considered each bit separately
- The only arithmetic component we saw was the simple counter, which incremented (or decremented, or reset) its unsigned integer "count value" one by one
- While such circuits are indeed a large application area for reconfigurable logic ICs, FPGAs are nowadays hailed for their abilities of parallel processing of custom arithmetics, mostly due to the AI hype and its endless hunger for more powerful compute
- In this part of the course, we'll start by defining arithmetics for FPGAs and discuss migrating the computational graphs we define on paper to the digital domain
- Then, we will dive into building useful applications with those

Outline

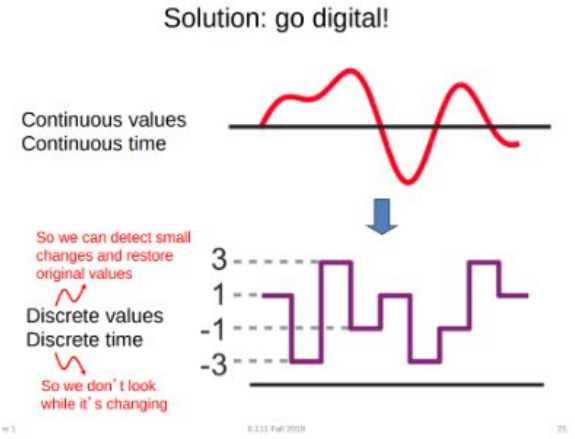
- Quantization
- Fixed point (FxP) vs. floating point
- FxP number formats and arithmetic operations

Outline

- Quantization
- Fixed point (FxP) vs. floating point
- FxP number formats and arithmetic operations

Quantization

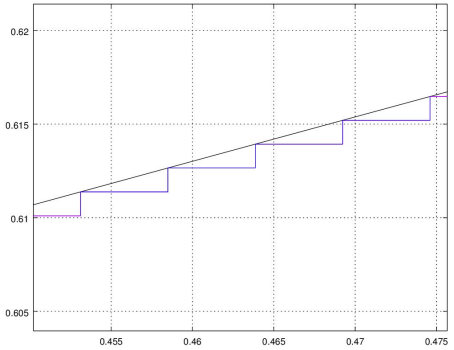
- Remember this analog to digital conversion slide from lecture 2? →→→
- digitization = sampling + quantization
- We mentioned that we would cover quantization in detail (for sampling, check, e.g., ELEC 303). We're at that point now.
- Basic definition of quantization: To get deterministic and repeatable outputs despite noise, we map sets of "real" values to discrete intervals. Larger intervals mean we need less resources but it also means more error w.r.t. to the "real" value, so we try to engineer these discrete intervals to minimize both resource usage and quantization error

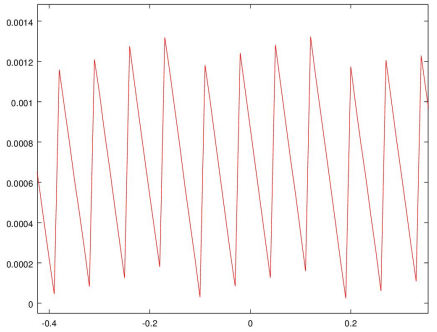


img src: https://web.mit.edu/6.111/volume2/www/f2019/handouts/L01.pdf

Quantization

- To migrate algorithms to the digital domain, we apply quantization to **every** number on the computational graph of that algorithm.
- Depending on the quantization scheme we choose, we will get different amounts of error on each number (compared to the “real value”), which will affect the accuracy of the algorithm
- The algorithms designer in a digital design team needs to take this into account while developing the computational graph (typically via tweaking some parameters)
- The error is non-deterministic (you never know what the real value is), so you design “around it” (e.g., if the max quantization error is too high for your specs, you increase the bitwidth etc.)





img src: https://github.com/dicearr/neuron-vhdl/tree/master

Quantization

- Let's analyze the computational graph of an example algorithm: linear regression

$$\mathbf{y} = \mathbf{X}\boldsymbol{\beta} + \boldsymbol{\varepsilon}$$

$$\mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix},$$

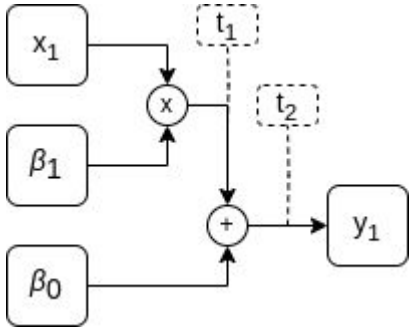
$$\mathbf{X} = \begin{bmatrix} \mathbf{x}_1^{\mathrm{T}} \\ \mathbf{x}_2^{\mathrm{T}} \\ \vdots \\ \mathbf{x}_n^{\mathrm{T}} \end{bmatrix} = \begin{bmatrix} 1 & x_{11} & \cdots & x_{1p} \\ 1 & x_{21} & \cdots & x_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n1} & \cdots & x_{np} \end{bmatrix}$$

$$\boldsymbol{\beta} = \begin{bmatrix} \beta_0 \\ \beta_1 \\ \beta_2 \\ \vdots \\ \beta_p \end{bmatrix}, \quad \boldsymbol{\varepsilon} = \begin{bmatrix} \varepsilon_1 \\ \varepsilon_2 \\ \vdots \\ \varepsilon_n \end{bmatrix}.$$

- For $n = p = 1$ (single unit, single parameter), the equation becomes:

$$y_1 = \beta_0 + x_1 * \beta_1$$

the computational graph for this algorithm looks like this:

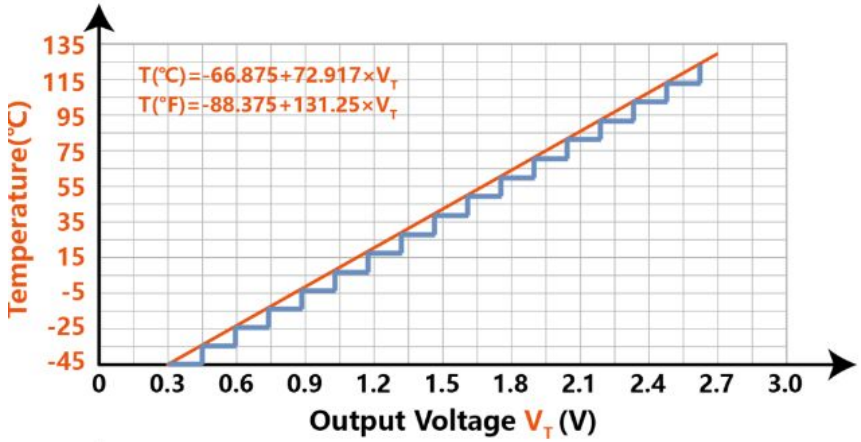


- Basically there are 6 numbers here : x_1 , y_1 , t_1 , t_2 , β_0 , β_1
- They have "real values" (in $\mathbb{R}^6$) which we need to quantize

Quantization

- Recall how we tackled this in lecture 2 with a single number, what we did was just one way of doing this. We'll now see alternative methods and their pros/cons and do this on a computational graph level (to all 6 numbers, considering their application requirements).

- Quantization → Let's chop this 0.3V-2.7V voltage range up to 32 pieces and represent it with a uniform fixed-point number representation (5-bits)



- This means we'll have the following values to work with (other values will be rounded to these somehow): {0.300, 0.375, 0.450, … , 2.550, 2.625}

  → Note how we don't have 2.7 anymore, that would require a 33rd value

- We can now treat this set of 32 values as a 5-bit digital value set (i.e., {b00000, b00001, b00010, …, b11110, b11111}) and design around it

Outline

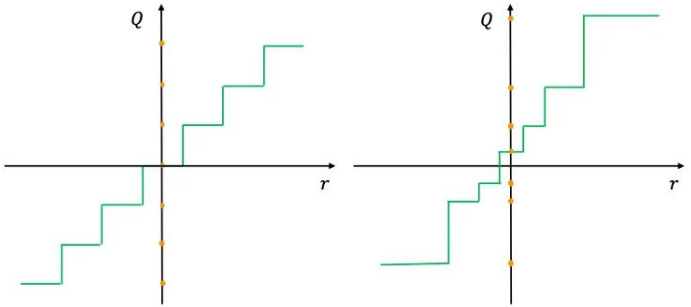
- Quantization
- Fixed point (FxP) vs. floating point
- FxP number formats and arithmetic operations

Outline

- Quantization
- Fixed point (FxP) vs. floating point
- FxP number formats and arithmetic operations

Fixed Point vs. Floating Point

- We make the following choices when quantizing our numbers:
  - uniform vs. non-uniform: regarding the distribution of the sizes of discretization intervals
  - scalar vs. vector: regarding whether numbers are quantized with a scheme individually, or are they grouped into vectors and quantized relatively. We will mostly cover scalar methods.
  - how is the “rounding” done: when we quantize the analog reading, which way do we snap the value? (ceil, floor, towards zero, nearest etc.)



img src:
https://medium.com/@florian_algo/model-quantization-2-uniform-and-non-uniform-quantization-47ca5b5d3ec0

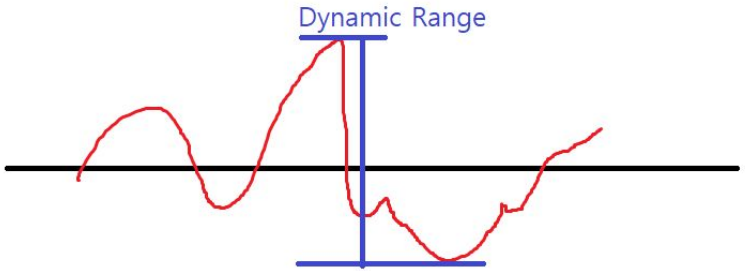
Fixed Point vs. Floating Point

- The uniform vs. non-uniform choice is typically made looking at the dynamic range (max-min) and precision (step size) requirements of the quantities that you're interested in.

- If you need a relatively small range of numbers with the same precision requirements all throughout the range, you should go with uniform quantization, specifically with fixed point.

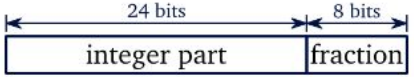
- If you need to cover a very large range, and you need high precision in the lower ranges but are OK with lower precision in the higher ranges, you can go with non-uniform quantization, specifically with floating point

- However don't let this difference trick you, you're still getting approximations of real numbers in both cases (in fractional form). Only difference is in how well that approximation is and "where" it is (on the number line)



img src: https://m.blog.naver.com/ptm0228/222086661956

Fixed Point vs. Floating Point



img src: http://www.cburch.com/books/float/

- Fixed point (FxP) number representations (uniform and scalar) are called so because they “affix the decimal point” (i.e., the fractional part of the number). They are defined by three parameters: 1) a scale factor, 2) a zero point mapping (bias), and 3) a rounding method

- You might have run into these in an image processing task with the name “normalization”, where you map a range of “real” numbers (actually they were floats), say *x*, in the range {0,1} to uint8 (8-bit, 256 different values) numbers in {0,255} to run imshow() on the result.

- Here, scale factor is 255, the zero point is 0.5 → 127.5, and arbitrary rounding can be chosen.

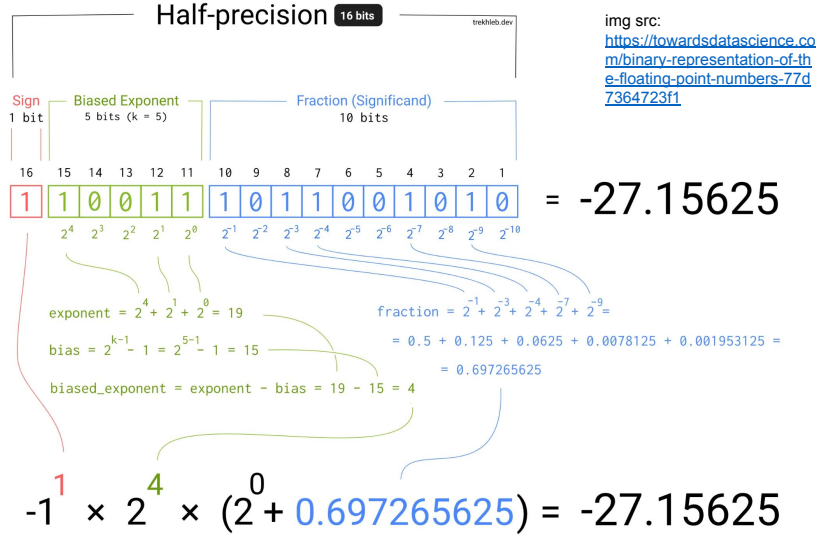
- The mapping is thus from {0, 0.00390625, 0.0078125, … , 0,99609375} to {0, 1, 2, …, 255}. Mark how this mapping does not have exact representations of the real values 1 (out of range) or 0.5 (since 127.5 cannot exist in the uint8 space, the closest are 127 and 128).

Fixed Point vs. Floating Point

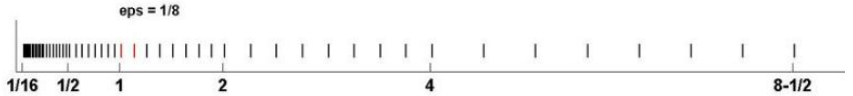
- To better picture this, we can think of FxP having an “internal representation” which the designer knows of, and a “represented value” which the circuit will operate on.
- The {0, 0.00390625, 0.0078125, … , 0,99609375} range is the internal representation, and the {0, 1, 2, …, 255} range is the values represented by them.
- One thing to note here before moving on is that every interval within those ranges have the same value (i.e., the number of “ticks” are the same for any given two intervals of the same size within the range). This is why FxP is **uniform quantization.**
- Furthermore, every value is quantized individually. This is why FxP is **scalar quantization.**
- If vector quantization was used, the scale factor and zero points would be determined based on groups within ranges (e.g., {0, 0.2} would have one factor, {0.2, 1.0} would have another)

Fixed Point vs. Floating Point

- Floating point is non-uniform (but still scalar quantization). The number of ticks in two intervals of the same size are different when they are at different points
- The closer you are to 0, the more resolution you get, and the farther away, the less.
- However, thanks to this we can represent very large numbers that would require many many many more bits to represent with FxP!
- A good read: *"What Every Computer Scientist Should Know About Floating-Point Arithmetic"*



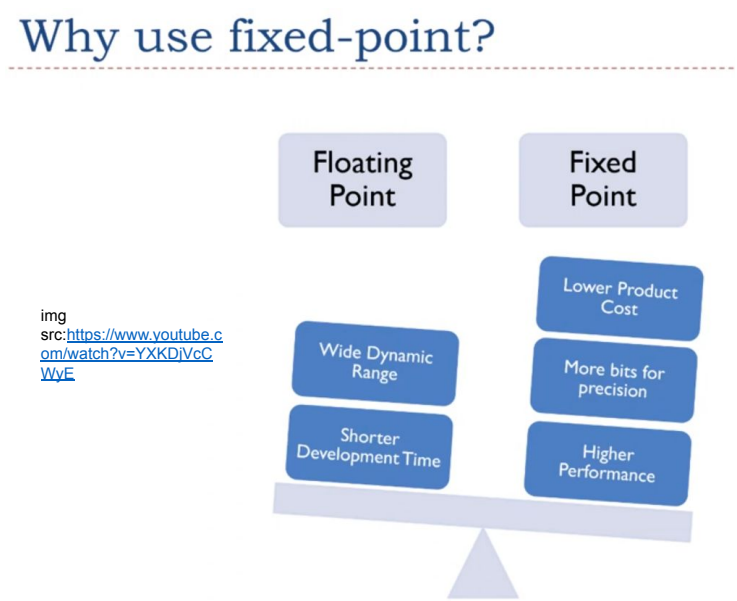
img src: https://towardsdatascience.com/binary-representation-of-the-floating-point-numbers-77d7364723f1



img src: https://blogs.mathworks.com/cleve/2014/07/07/floating-point-numbers/

Fixed Point vs. Floating Point

- Which one should we use?
- Like we said earlier: This depends on the application requirements, but the trade-off looks like this →→
- There is one additional consideration for us since we're designing hardware: If we are going to use floating point numbers, we need to have the circuitry that can do arithmetic with them!
- The arithmetic operators we saw up to now (full adders etc.) worked on integers, they didn't consider decimal points and fractions.
- These components can be used with FxP, but floating point numbers need special circuitry



img src:https://www.youtube.com/watch?v=YXKDjVcCWyE

- We will focus on FxP representations for the rest of this course

Outline

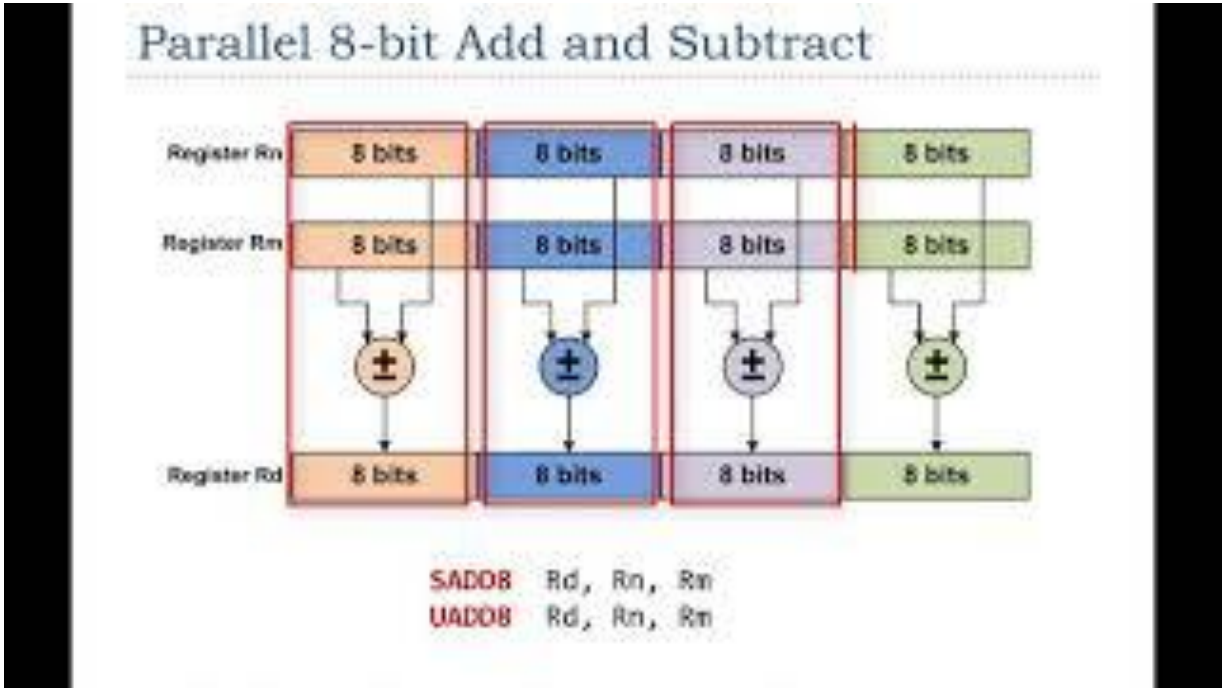
- Quantization
- Fixed point (FxP) vs. floating point
- FxP number formats and arithmetic operations

Outline

- Quantization
- Fixed point (FxP) vs. floating point
- FxP number formats and arithmetic operations

Fixed Point Number Formats and Arithmetic

- Let’s discuss this over Dr. Zhu’s slides:



next ⟶ DSP + optimization