Introductory course to VHDL and HLS FPGA programming
Day 4

Sioni Summers (CERN)
26 June 2025 - Milan

Introduction

- High Level Synthesis is a new paradigm in programming FPGAs
  - Write algorithms ➝ synthesis tool determines the hardware
  - Input using C++ — higher level abstraction than HDLs
  - Productivity 📈 — create working designs faster
  - Sophistication 📈 — create advanced designs with complicated algorithms
- But working with HLS still requires expertise, and a foundation in HDL is a great starting point
- Main topics of this part:
  - Number representations and arithmetic
  - Loop Analysis and Optimization

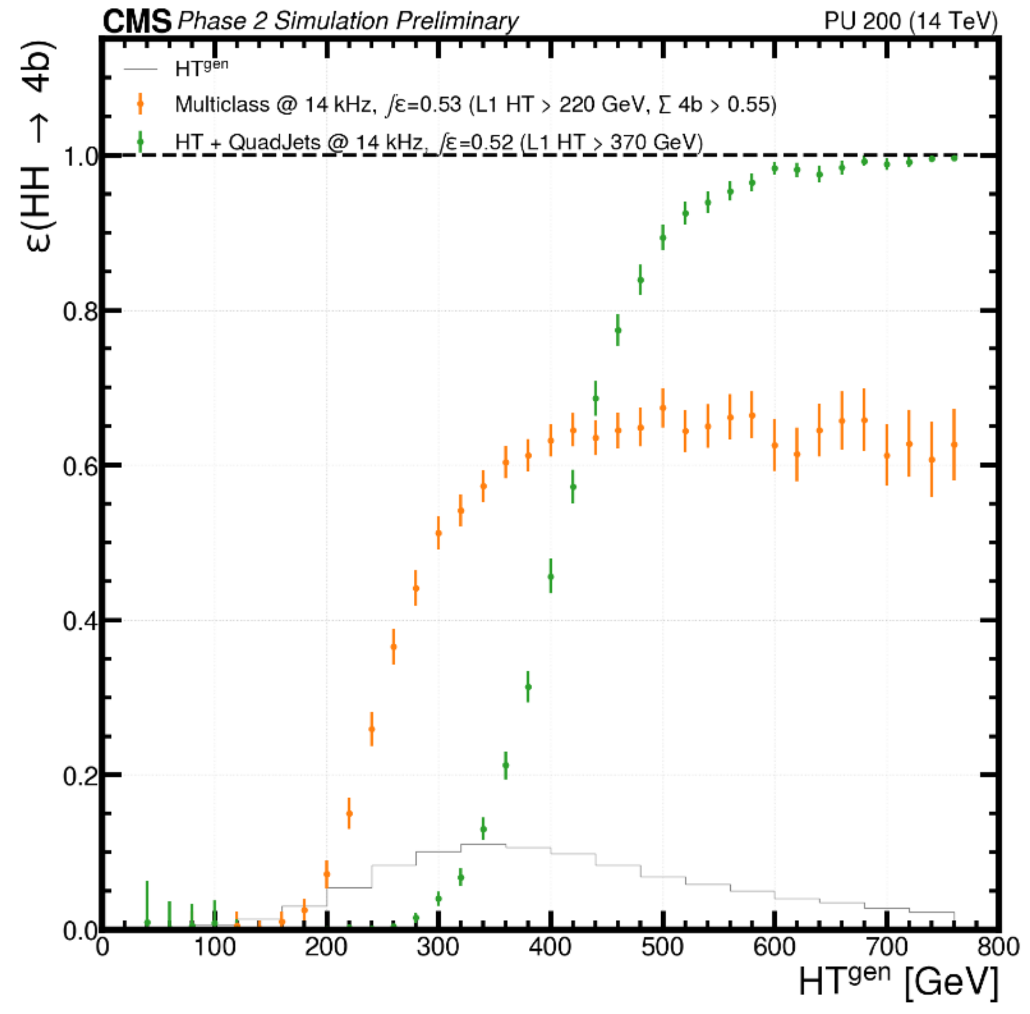
About me

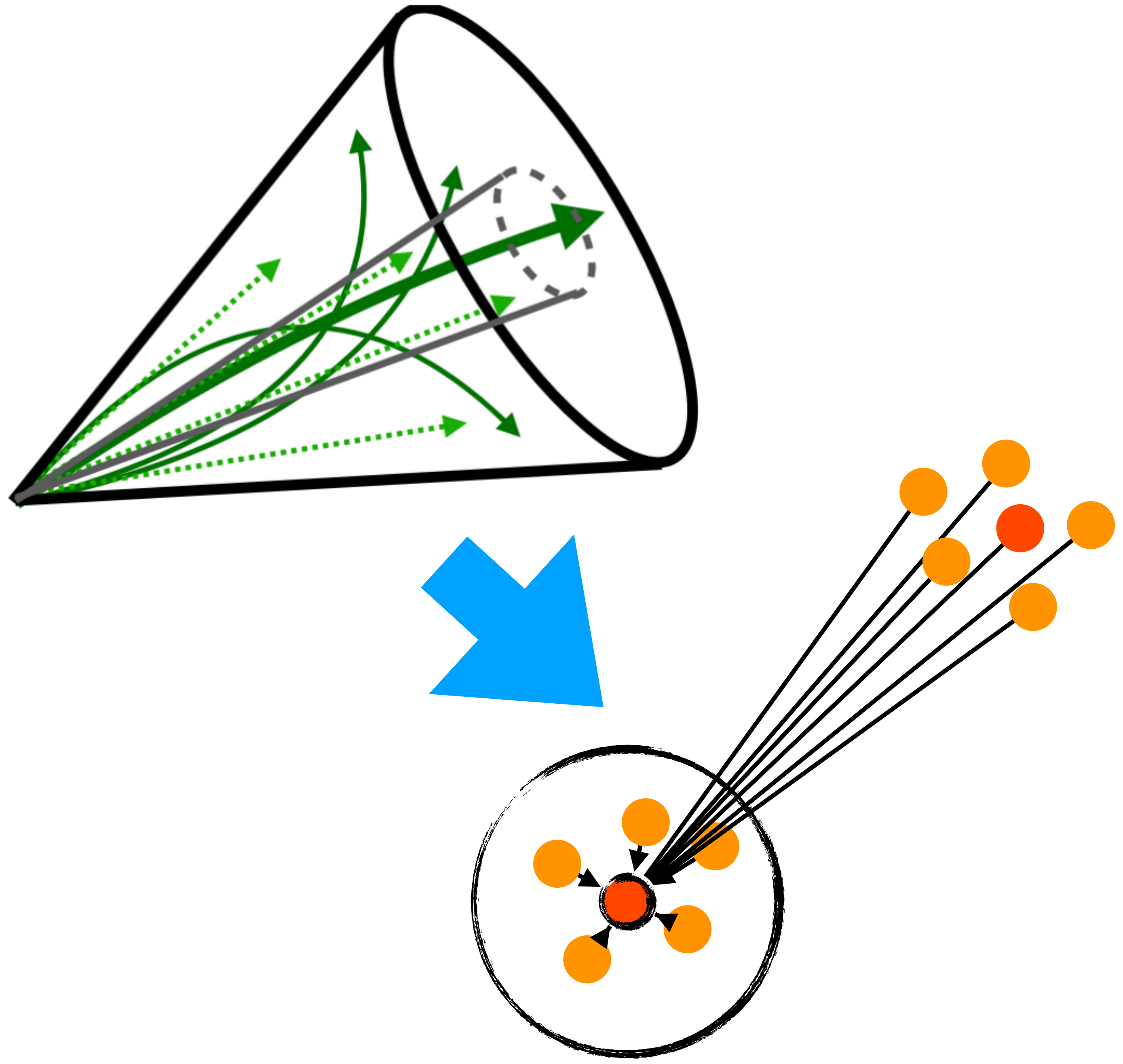
- Staff at CERN working on Level 1 Trigger Upgrade for CMS experiment
  - Mostly designing and implementing detector reconstruction algorithms for Level 1 Trigger
  - Track reconstruction, vertexing, particle flow, jets, jet tagging, ML
  - Task leader in Next Generation Triggers project
- PhD High Energy Physics Imperial College London
  - Thesis: “Applications of FPGAs to triggering in particle physics”
  - Designing physics algorithms with high level languages for FPGAs
- Also deploying Machine Learning into FPGAs for low latency
  - **hls4ml** coordinator 2020-2022, creator and maintainer of **conifer**
- Leading Edge SpAIce project at CERN: ML in FPGA for satellites

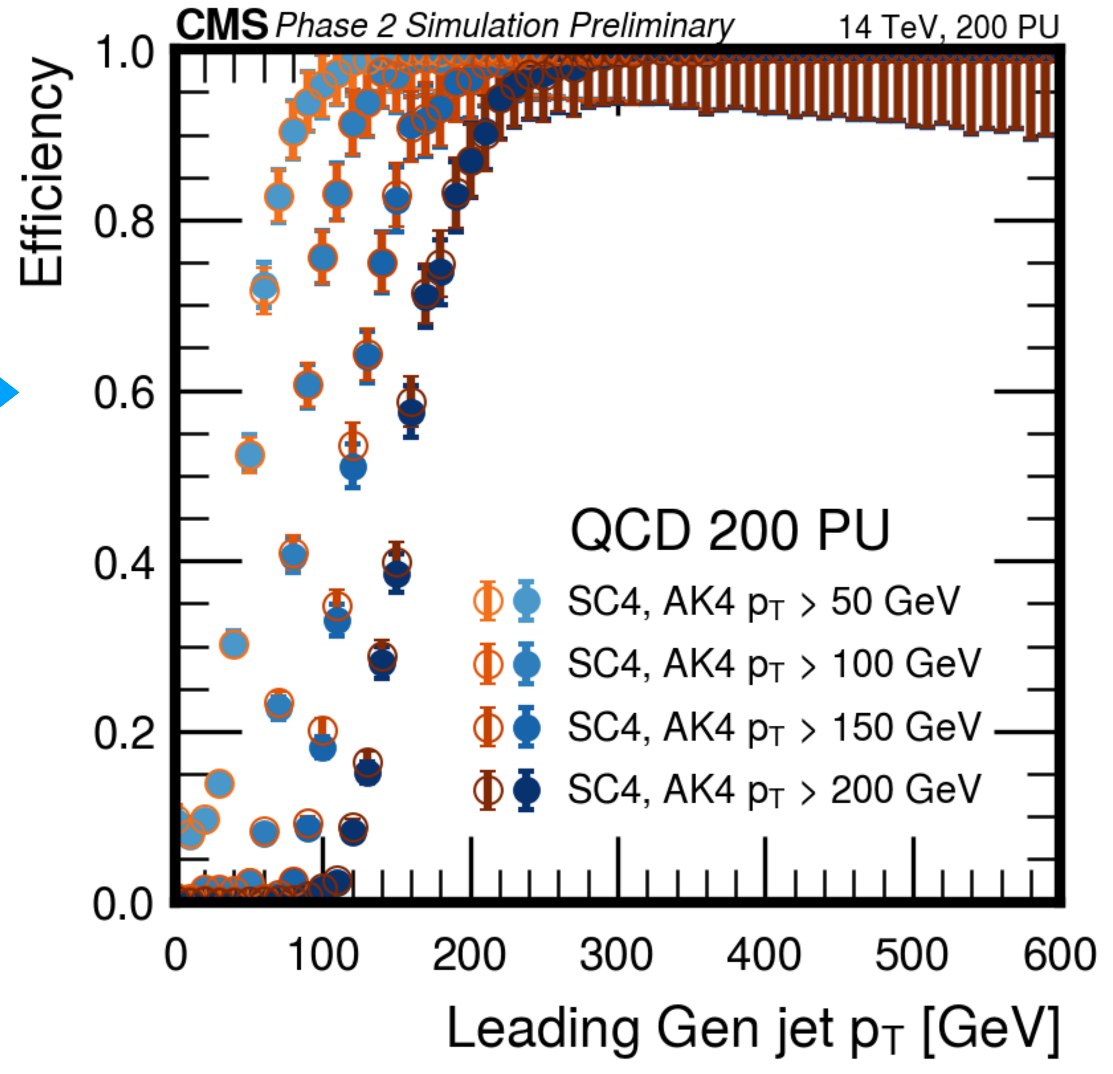
sioni@cern.ch
sioni.web.cern.ch
@thesps
@ssummers

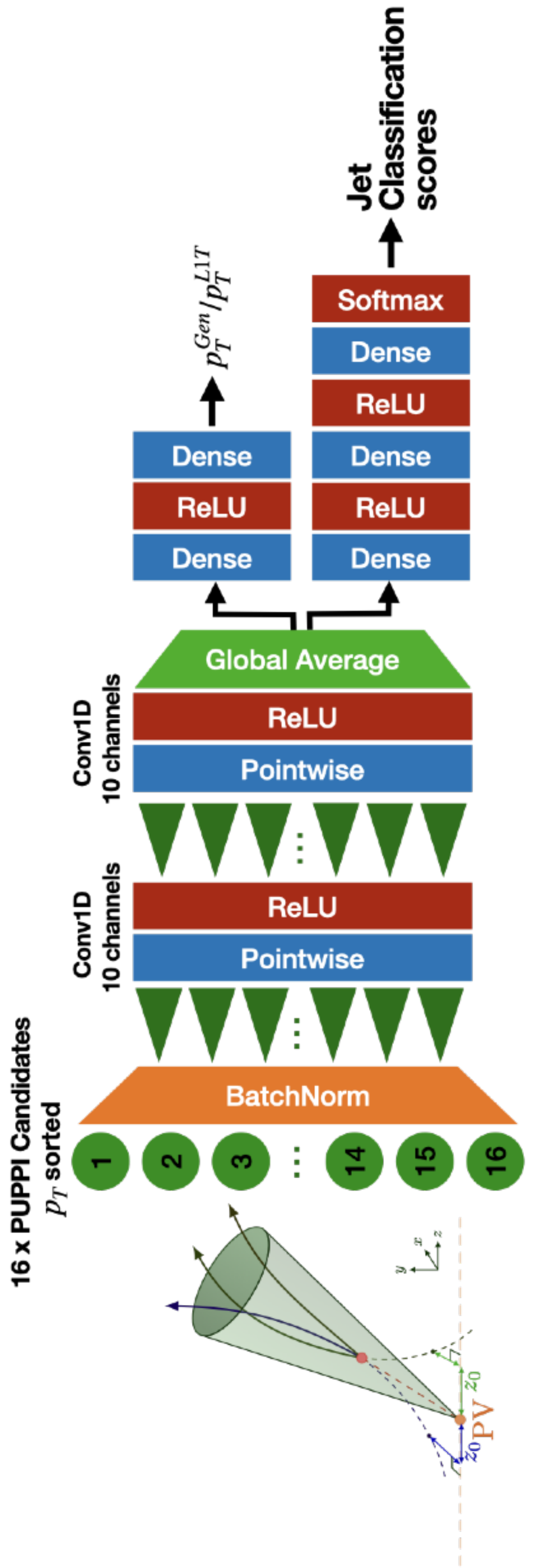
One FPGA Case Study

- For the CMS Phase 2 Upgrade we are designing jet reconstruction and tagging
    - Find cones of particles from the decay of a parent particle, and use ML to predict the parent particle type, make CMS will collect data for important event types (e.g. HH➞bbbb)
- Less than 1 μs from particles input to tagged jets output
- We used both HLS and VHDL, and **hls4ml** for the Neural Network

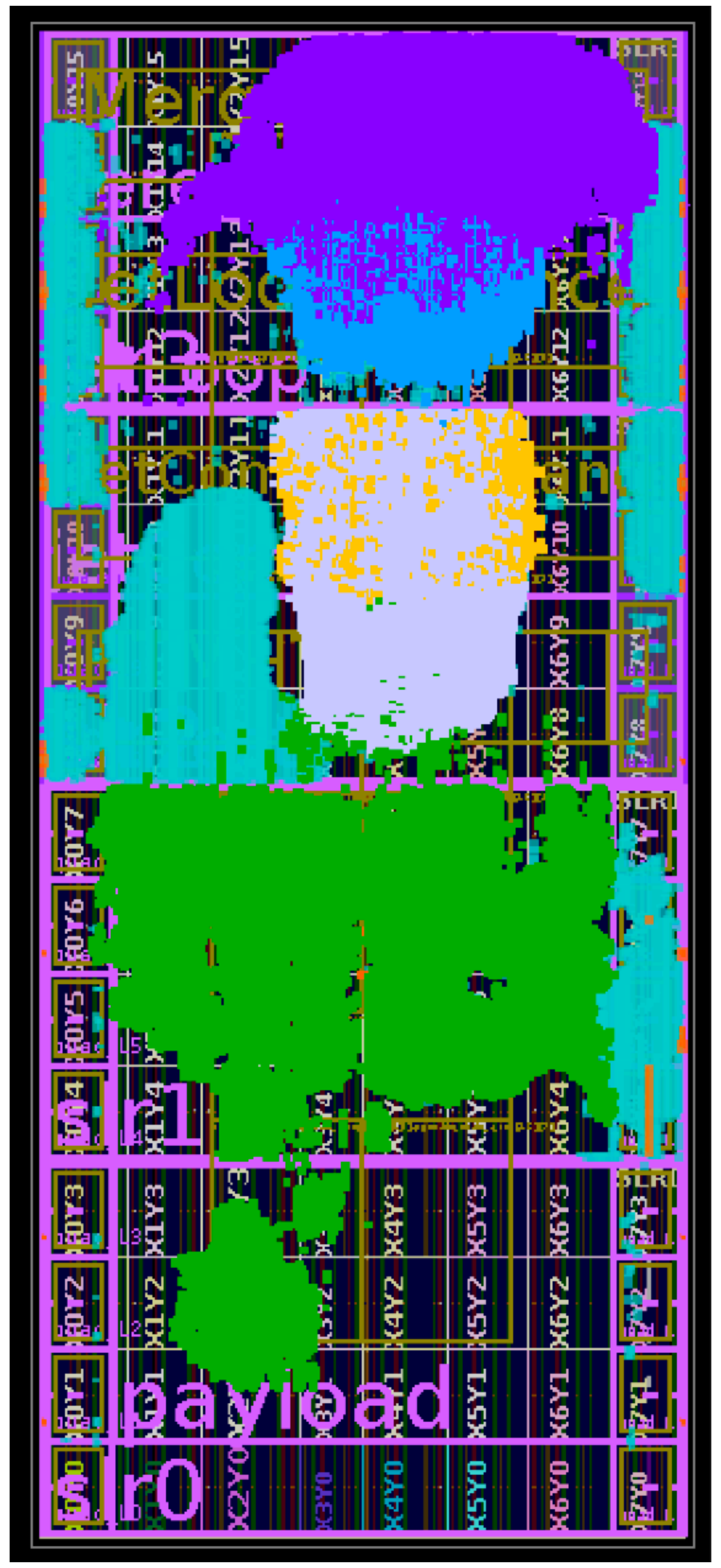








Particle receiving
Jet finding
Jet tagging (NN)



Part 1

Numerics: Fixed Point Arithmetic

- On your CPU (and GPU) you are probably familiar with integer and floating point numerical formats
- Integer represents integer values in a fixed number of bits (e.g. 32, 16, 8)
- Floating point represents values with a mantissa and exponent : $m \cdot 2^e$
  - It's like scientific notation with binary
- Simplified examples, representing the number 113, ignoring sign (positive values only)

Base 10 integer
113_{10}

10^2	10^1	10^0
1	1	3

Base 10 floating point
(3 digit mantissa,
2 digit exponent)
$1.13 \times 10^{2.0}$

10^0		10^{-1}	10^{-2}
1	.	1	3

$\cdot\ 10$^

10^0		10^{-1}
2	.	0

Base 2 integer
0b01110001

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
0	1	1	1	0	0	0	1

Base 2 floating point
(4 bit mantissa,
4 bit exponent)
$14 \times 2^8 = 112$
(Approximate)

2^3	2^2	2^1	2^0
1	1	1	0

$\cdot\ 2$^

2^3	2^2	2^1	2^0
1	0	0	0

Operations

- On your CPU (and GPU) you are probably familiar with integer and floating point numerical formats
- Integer represents integer values in a fixed number of bits (e.g. 32, 16, 8)
- Floating point represents values with a mantissa and exponent : $m \cdot 2^e$

- Demo:
  - Suppose we have decimal integers `x = 15` and `y = 27`
  - Compute `z = x + y` using “long hand”

X	1	5
Y	2	7
Carry	1	
Z	4	2

Integers

- On your CPU (and GPU) you are probably familiar with integer and floating point numerical formats
- Integer represents integer values in a fixed number of bits (e.g. 32, 16, 8)
- Floating point represents values with a mantissa and exponent : $m \cdot 2^e$

- **Exercise**:
  - Suppose we have 4-bit unsigned (positive) integers `x = 0b0011` and `y = 0b0101` ← Note: 0b prefix means binary number
  - Compute `z = x + y` — use the "long hand" method and remember to "carry the 1"

Answer

Integer addition

- On your CPU (and GPU) you are probably familiar with integer and floating point numerical formats
- Integer represents integer values in a fixed number of bits (e.g. 32, 16, 8)
- Floating point represents values with a mantissa and exponent : $m \cdot 2^e$

- **Exercise**:
  - Suppose we have 4-bit unsigned (positive) integers `x = 0b0011` and `y = 0b0101`
  - Compute `z = x + y` — use the “long hand” method and remember to “carry the 1”

Note: 0b prefix means binary number

x	0	0	1	1
y	0	1	0	1
	1	1	1	
Z	1	0	0	0

Sanity check:
$0b0011 = 3_{10}$
$0b0101 = 5_{10}$
$3 + 5 = 8$
$8_{10} = 0b1000$
✓

Note: subscript$_{10}$ means decimal number

Overflow, saturation, truncation

- When working with decimals we usually perform "bit growth" intuitively e.g. `9 + 3 = 12` — one more digit in result
- With computer arithmetic the bit sizes are usually fixed e.g. 4, 8, 16, 32 bits
- When results of operations exceed the constraints of the precision, we can get *overflow*
  - **Exercise**: compute $12_{10} + 5_{10}$ with 4 bit operands and 4 bit result ie ignore anything beyond 4 bits

Answer

Overflow, saturation, truncation

- When working with decimals we usually perform "bit growth" intuitively e.g. `9 + 3 = 12` — one more digit in result
- With computer arithmetic the bit sizes are usually fixed e.g. 4, 8, 16, 32 bits
- When results of operations exceed the constraints of the precision, we can get *overflow*
  - **Exercise**: compute $12_{10} + 5_{10}$ with 4 bit operands and 4 bit result ie ignore anything beyond 4 bits

x		1	1	0	0
y		0	1	0	1
Z	~~1~~	0	0	0	1

- Overflows can be problematic, data is essentially corrupted
  - $12_{10} + 5_{10} = 1_{10}$?
  - Results of summing two positive values could be negative

Answer

Overflow, saturation, truncation

- When working with decimals we usually perform “bit growth” intuitively e.g. `9 + 3 = 12` — one more digit in result
- With computer arithmetic the bit sizes are usually fixed e.g. 4, 8, 16, 32 bits
- When results of operations exceed the constraints of the precision, we can get *overflow*
    - **Exercise**: compute $12_{10} + 5_{10}$ with 4 bit operands and 4 bit result ie ignore anything beyond 4 bits

x		1	1	0	0
y		0	1	0	1
	1	1			
Z	~~1~~	0	0	0	1

- Overflows can be problematic, data is essentially corrupted
- We can *saturate* to avoid overflows — the result is still “wrong” but is likely to be more useful than the overflowed one
    - Clip the value to the largest (or most negative) value
- **Exercise**: what would be the saturated value of the previous exercise?

Bit Growth

- When results of operations exceed the constraints of the precision, we can get *overflow*
- With computer arithmetic the bit sizes are usually fixed e.g. 4, 8, 16, 32 bits
- We can increase the bit precision of the result of an operation in an FPGA to compensate for overflow
    - Provided the result is stored in a different memory/register/logic than the operands of smaller width

- **Exercise**: assuming unsigned integers, what would be the required bit-width for the result of:
    - a 4-bit integer summed with a 4-bit integer?
    - a 4-bit integer summed with a 3-bit integer?
    - a N-bit integer summed with an M-bit integer?
    - a 4-bit integer multiplied with a 4-bit integer?
    - a 4-bit integer multiplied with a 3-bit integer?
    - a N-bit integer multiplied with an M-bit integer?

Hint: consider the maximum values of each operand

Answer

Bit Growth

- **Exercise**: assuming unsigned integers, what would be the required bit-width for the result of:
  - a 4-bit integer summed with a 4-bit integer?
  - a 4-bit integer summed with a 3-bit integer?
  - a N-bit integer summed with an M-bit integer?
  - a 4-bit integer multiplied with a 4-bit integer?
  - a 4-bit integer multiplied with a 3-bit integer?
  - a N-bit integer multiplied with an M-bit integer?

- General rules to guarantee no overflow:
  - for addition & subtraction the result bitwidth should be 1 + max(N,M)
  - for multiplication the result bitwidth should be N + M

Two's complement

- So far we used *unsigned* integers, but that's limiting
- How do we represent negative values with fixed size binary numbers?
- Most common method: two's complement
- To work out the two's complement representation of a negative number (e.g. -6_{10} in 4 bits)
  - Start with the binary representation of the *absolute* value: $\mathtt{6_{10} = 0b0110}$
  - *Invert* all of the bits: `0b0110 → 0b1001`
  - Add 1 to the value, *ignoring* overflow: `0b1001 → 0b1010`
- Observations:
  - The most significant bit always denotes the sign — leading 0 ➞ positive or zero, leading 1 ➞ negative (but not sign & value)
  - We can do arithmetic with numbers in this representation
  - **Exercise**: take two 4 bit two's complement numbers $\mathtt{x = +3_{10}}$, $\mathtt{y = -1_{10}}$ and compute `x + y` in binary

Answer

Two's complement exercise

- **Exercise**: take two 4 bit two's complement numbers `x = +3`$_{10}$, `y = -1`$_{10}$ and compute `x + y` in binary

x		0	0	1	1
y		1	1	1	1
	~~1~~	1	1	1	
Z		0	0	1	0

Sanity check:
0b0011 = 3_{10}
0b1111 = -1_{10}
3 - 1 = 2
2_{10} = 0b0010
✓

- Exercise: what are the maximum and minimum values of a:
  - 4 bit unsigned integer
  - 4 bit two's complement integer
  - 7 bit two's complement integer

Two’s complement exercise

- **Exercise**: take two 4 bit two’s complement numbers `x = +3`$_{10}$, `y = -1`$_{10}$ and compute `x + y` in binary

x		0	0	1	1
y		1	1	1	1
	~~1~~	1	1	1	
Z		0	0	1	0

Sanity check:
0b0011 = 3_{10}
0b1111 = -1_{10}
3 - 1 = 2
2_{10} = 0b0010
✓

- **Exercise**: what are the maximum and minimum values (excluding zero) of a:
  - 4 bit unsigned integer — Max = 15, Min = 1
  - 4 bit two’s complement integer — Max = 7, Min = -8
  - 7 bit two’s complement integer — Max = 63, Min = -64

Fixed Point

- On CPU / GPU we need to work with number types that are native to the hardware
  - We can emulate other number types but it arithmetic won't run with high performance
- On FPGA we are designing the hardware itself, so we can use any number representation that we like
- We'll see this for ourselves soon, but integer operations are much less resource and latency intensive than floating point
- But what if we want to represent fractions? Enter fixed point
- Fixed point combines some of the convenience of floating point with the low hardware cost of integers
- Recall: floating point is $m \cdot 2^e$ → in fixed point the value of exponent is fixed so it doesn't need to be explicitly represented
- Example: 8 bit unsigned fixed point with 4 integer bits, 4 fractional bits, representing 9.3125_{10}

Integer					**Fraction**			
2^3	2^2	2^1	2^0	.	2^{-1}	2^{-2}	2^{-3}	2^{-4}
1	0	0	1	.	0	1	0	1

Binary point / radix point

Fixed Point

- **Exercise**: what are the values in base 10 of these 8 bit unsigned fixed-point numbers?
  - What are the maximum and minimum values of the three fixed-point number formats?

a.

Integer				Fraction			
2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}
0	1	1	1	0	0	0	1

b.

Integer						Fraction	
2^5	2^4	2^3	2^2	2^1	2^0	2^{-1}	2^{-2}
0	1	1	1	0	0	0	1

c.

Fraction							
2^{-4}	2^{-5}	2^{-6}	2^{-7}	2^{-8}	2^{-9}	2^{-11}	2^{-11}
0	1	1	1	0	0	0	1

Example

Integer				Fraction			
2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}
1	0	0	1	0	1	0	1

$= 9.3125_{10}$

Answer

Fixed Point

- **Exercise**: what are the values in base 10 of these 8 bit unsigned fixed-point numbers?
  - What are the maximum and minimum values of the three fixed-point number formats?

a.

Integer				Fraction			
2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}
0	1	1	1	0	0	0	1

X = 7.0625
Max = 15.9375
Min = 0.0625

b.

Integer						Fraction	
2^5	2^4	2^3	2^2	2^1	2^0	2^{-1}	2^{-2}
0	1	1	1	0	0	0	1

x = 28.25
Max = 63.75
Min = 0.25

c.

Fraction							
2^{-4}	2^{-5}	2^{-6}	2^{-7}	2^{-8}	2^{-9}	2^{-11}	2^{-11}
0	1	1	1	0	0	0	1

X = 0.05517578125
Max = 0.12451171875
Min = 0.00048828125

Example

Integer				Fraction			
2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}
1	0	0	1	0	1	0	1

$= 9.3125_{10}$

Fixed Point

- **Exercise**: can you generalise the rules for bit-growth of integers to bit-growth of fixed-point?
  - **1)** For addition/subtraction and **2)** multiplication. **Hint**: consider maximum and minimum values
  - Operand 0: F_0 fractional bits, I_0 integer bits (F_0 + I_0 width); Operand 1: F_1 fractional bits, I_1 integer bits (F_1 + I_1 width)
  - Recall the integer bit-growth rules, with operand widths 'N' and 'M'
    - for addition & subtraction the result bitwidth should be 1 + max(N,M)
    - for multiplication the result bitwidth should be N + M

Answer

Fixed Point

- **Exercise**: can you generalise the rules for bit-growth of integers to bit-growth of fixed-point?
  - **1)** For addition/subtraction and **2)** multiplication. **Hint**: consider maximum and minimum values
  - Operand 0: F_0 fractional bits, I_0 integer bits ($F_0 + I_0$ width); Operand 1: F_1 fractional bits, I_1 integer bits ($F_1 + I_1$ width)
  - Recall the integer bit-growth rules, with operand widths 'N' and 'M'
    - for addition & subtraction the result bitwidth should be 1 + max(N,M)
    - for multiplication the result bitwidth should be N + M
- Addition:
  - To safely add/subtract two fixed-point values, align the binary point first (HLS will do this automatically)
  - The result must accommodate the largest integer range and maximum fractional precision
  - Integer bits : $\max(I_0, I_1) + 1$ // +1 for carry
  - Fractional bits: $\max(F_0, F_1)$
  - Total width : $\max(I_0, I_1) + \max(F_0, F_1) + 1$

Answer

Fixed Point

- **Exercise**: can you generalise the rules for bit-growth of integers to bit-growth of fixed-point?
  - **1)** For addition/subtraction and **2)** multiplication. **Hint**: consider maximum and minimum values
  - Operand 0: F_0 fractional bits, I_0 integer bits ($F_0 + I_0$ width); Operand 1: F_1 fractional bits, I_1 integer bits ($F_1 + I_1$ width)
  - Recall the integer bit-growth rules, with operand widths 'N' and 'M'
    - for addition & subtraction the result bitwidth should be 1 + max(N,M)
    - for multiplication the result bitwidth should be N + M
- Multiplication:
  - When multiplying two fixed-point numbers, both integer and fractional parts grow
  - The binary point is at the sum of the original positions
  - Integer bits : $I_0 + I_1$
  - Fractional bits: $F_0 + F_1$
  - Total width : $I_0 + I_1 + F_0 + F_1$

Introduction to High Level Synthesis

- Now we'll get some hands on experience with numerics in High Level Synthesis
- Reminder: why do we use it?
  - Write FPGA designs in C++: lower barrier to entry than HDL
  - Rapid design space exploration (explore resource / latency tradeoffs with ease); realise complex algorithms
- 🛠️ Typical HLS workflow
  - ✏️ Write C/C++ *kernel*: my_function.cpp
  - 🧪 Create *testbench*: my_function_tb.cpp to simulate inputs/outputs
  - ⚙️ Run HLS flow using Vitis HLS:
    - C Simulation: functional correctness check using the testbench — HLS C++ code is executed on the CPU
    - C Synthesis: HLS C++ code is *synthesized* into RTL + resource/latency estimates
    - Co-Simulation: validate generated RTL against testbench — clock cycle accurate
- Optimize latency, throughput, and resource usage (LUTs, FFs, DSPs, BRAM)
  - Explore trade-offs using loop unrolling, pipelining, precision tuning

Introduction to High Level Synthesis

- 📁 Structure of a Typical HLS Project
  - 🔧 my_function.cpp — algorithm to be synthesized to FPGA
  - 📊 my_function_tb.cpp — C++ testbench that drives inputs and checks outputs
  - 📂 hls_prj/ — directory with reports, logs, RTL, etc. produced from the HLS tool
  - 🧾 script.tcl — automates synthesis/verification in batch mode

```
open_project my_proj
add_files my_function.cpp
add_files -tb my_function_tb.cpp
open_solution "solution1"
set_top my_function
create_clock -period 5
csim_design
csynth_design
cosim_design
```

Introduction to High Level Synthesis

- 🔧 my_function.cpp — algorithm to be synthesized to FPGA
  → This is where you describe your logic in standard C++ (with some HLS-specific types)

- 🎯 Define a “top-level” function
  → This is the function that Vitis HLS treats as the hardware module interface
  → Must use scalar or array arguments (no dynamic memory, no STL containers)

- 🧮 Use ap_fixed datatypes for fixed-point arithmetic
  → Provided by the ap_fixed.h header
  → Enables precise control of bit widths for resource and accuracy trade-offs

  - 🔢 **`ap_fixed<16, 5>`**
    → 16-bit signed number: 5 integer bits, 11 fractional bits

  - 🔢 **`ap_fixed<8, 3, AP_RND, AP_SAT>`**
    → 8-bit signed, 3 integer bits, 5 fractional, AP_RND = round to nearest; AP_SAT = saturate on overflow

- 🏗️ Use **`#pragma`** HLS directives to control synthesis behavior
  → Guide unrolling, pipelining, array partitioning, and interface behavior
  → These affect latency, throughput, and resource usage

- 📅 Tomorrow: we'll explore how to use pragmas to tune designs for performance and resource efficiency

Introduction to High Level Synthesis

- Example HLS top-level function, sum two dimension-8 vectors of 16 bit, 8 integer bit values

```
#include "ap_fixed.h"

typedef ap_fixed<16,8> data_t;

void vec_sum(const data_t in1[8], const data_t in2[8], data_t out[8]) {
    VectorLoop:
    for (int i = 0; i < 8; ++i) {
        out[i] = in1[i] + in2[i];
    }
}
```

- When you have all of the ingredients, launch the workflow from the command line with:
    - **`vitis_hls -f csim.tcl ; vitis_hls -f csynth.tcl ; vitis_hls -f cosim.tcl`**
    - This will create a project, run simulation, synthesis, and/or co-simulation as defined in the script

Analyzing Designs

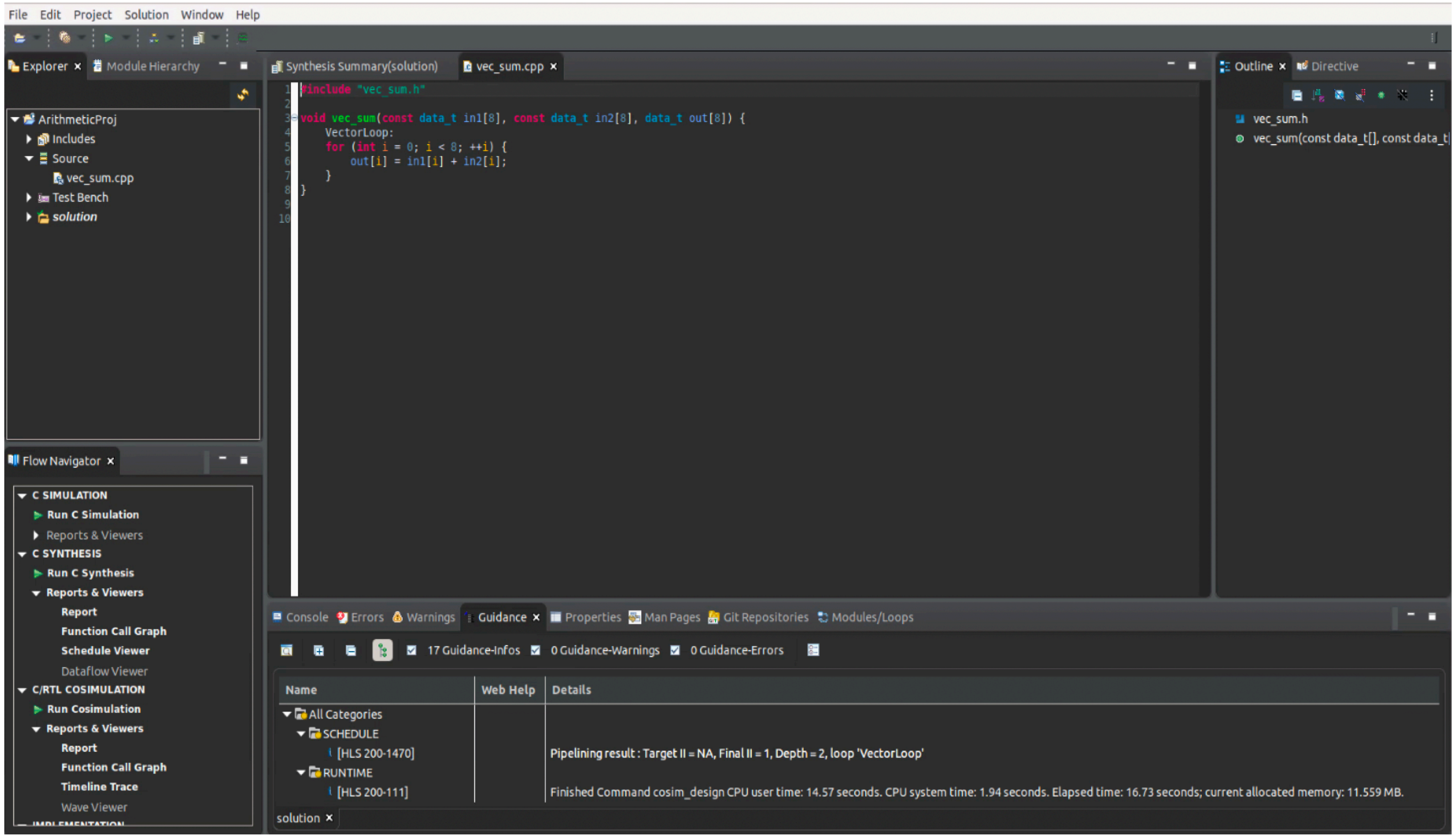
- After we run C Synthesis and Co Simulation, we generate many reports and also an HLS project
  - All of them contain very useful information for analyzing the design
  - In particular:
    - <project name>/<solution name>/syn/report/csynth.rpt
    - <project name>/<solution name>/sim/report/<top name>_cosim.rpt
  - These reports can also be viewed in the GUI (next slides)

```
+ Performance & Resource Estimates:

    PS: '+' for module; 'o' for loop; '*' for dataflow
    +---------------+------+------+---------+---------+----------+---------+------+----------+------+----+-----------+-----------+-----+
    |    Modules    | Issue|      | Latency | Latency | Iteration|         | Trip |          |      |    |           |           |     |
    |    & Loops    | Type | Slack| (cycles)|   (ns)  |  Latency | Interval| Count| Pipelined| BRAM | DSP|     FF    |    LUT    | URAM|
    +---------------+------+------+---------+---------+----------+---------+------+----------+------+----+-----------+-----------+-----+
    |+ vec_sum      |     -|  0.87|       XX|  XXX.XXX|        XX|       XX|    XX|       y/n|    XX|  XX|         XX|         XX|   XX|
    | o VectorLoop  |     -|  7.30|        X|  XXX.XXX|        XX|       XX|    XX|       y/n|    XX|  XX|         XX|         XX|   XX|
    +---------------+------+------+---------+---------+----------+---------+------+----------+------+----+-----------+-----------+-----+
```

HLS GUI

- We will also use the HLS GUI for some analysis, to open: run **`vitis_hls -classic`** (classic option for ≥ 2023.2)
- You may use whichever file editor you like, but the HLS GUI does provide some useful auto-completion



HLS GUI

- The analysis after running synthesis is especially useful e.g. schedule viewer
  - did the design map to HW as expected? Where are the bottlenecks in the design impacting performance?