

# Introduction to HLS

---

Simone Bologna

*simone.bologna@bristol.ac.uk*

*University of Bristol*

23 October 2019

- Introduction to FPGA, VHDL, and HLS
- Getting started with HLS
- Life of a toy project from conception to (almost) implementation
- Tips and tricks
- Using C++ constructs in Vivado HLS

# Introduction

---

- FPGA are circuits that are programmable on the field
- FPGAs are powerful and flexible devices
- Components of FPGA
  - Flip-Flops (FF), small memory component able to store a bit
    - Typical used as a fast register to store data
  - Look-Up Tables (LUT), small memories used to store truth tables and perform logic functions
    - Typically used to perform operation such as “and”, “or”, sums or subtractions
  - Digital Signal Processor (DSP), small processor able to quickly perform mathematical operation on streaming digital signals
    - Typically used for multiplication and additions
  - Block RAM (BRAM), memory able to store data
    - Can store a fair amount of data, but slow and with a limited number of ports limiting memory throughput

- What is VHDL?
  - VHSIC Hardware Design Language
    - Very High Speed Integrated Circuit Hardware Description Language
      - ... ergh...
  - Used to describe circuits that will be implemented on FPGA via code
  - Not covered here!
- High-Level Synthesis (HLS) enables user to transform (synthesise) C/C++/SystemC code into VHDL
  - Enables users to program FPGA in high-level languages!
  - Focusing on C++
- Analogies with assembly and high-level languages are stretched
  - Each language works better in specific situations

- Collecting here opinions I have heard from various experts
- When to use VHDL?
  - When you want full control on how your design is going to be implemented
  - When you need some clock-dependent applications
    - i.e. receive data and hold it for three clock cycle
  - Receiving data and sorting in specific manners
- When to use HLS?
  - Rapid prototyping
    - I would suggest to use it in doubt, implementing stuff in HLS will generally take less time than using VHDL
  - Designing some processing/analysis block
    - i.e. developing some particle identification algorithm

# Getting started

---

- `excession.phy.bris.ac.uk` is the FPGA development machine
- Two strategies to develop in HLS:
  - Write code in your favourite editor and use Vivado HLS' command line interface (CLI)
  - Use Vivado HLS's GUI to do both editing and synthesis
- Vivado HLS' command line does not provide all the tools
  - Vivado HLS GUI is required when you need to investigate design performance in detail
- Using editor + Vivado HLS CLI [here](#)
- I recommend using VNC to log into excession if you want to use Vivado GUI
  - Feel free to ask me help to set it up :)



- Get an account on excession
- Add Vivado to your environment
  - `source /software/CAD/Xilinx/2018.2/Vivado/2018.2/settings64.sh`
  - 2019.1 is available, I started on 2018.2 and I am keeping it for consistency
- Run **vivado\_hls** in your terminal to open the vivado\_hls GUI
  - Use if you have mounted /software locally or if you are working via VNC
- **vivado\_hls -i** opens the interactive TCL shell
  - Development tools through command line
- **vivado\_hls <.tcl file>** runs a .tcl script
  - Typically I use it to build my firmware and test
    - Back to tcl in a sec
- **vivado\_hls -f <.tcl file>** runs a .tcl script and keeps the console open
  - Useful for .tcl scripts that sets up your project before running some interactive operation

- **HLS file**, C/C++ code that will be synthesised and run on FPGA
- **Test bench (TB) file**, C/C++ code that is run to test the HLS code. It calls the HLS functions and can run tests on their output, e.g. C asserts.
- **Tcl scripts**, set of tcl instructions executed by the Vivado HLS shell
- **Synthesis**, C/C++ → HDL lang (VHDL/Verilog)
- **Project**, collection of HLS and test bench (TB) files
  - Has a top-level function name that is the starting point for synthesis
- **Solution**, specific implementation of a project
  - Runs on a specific device at a specific clock frequency
- **C simulation**, HLS + TB files are compiled with gcc against HLS headers and lib and plainly run as any other executable
- **C/RTL cosimulation**, synthesised HLS code is run on a simulator and results tested on the C/C++ test bench

- In a base project you will typically have
  - At least a HLS .c/.cpp files
  - A header used to link HLS code to test bench code
  - At least a TB .c/.cpp file
  - A .tcl script to set up your Vivado HLS project and solution

- Problem
- Define your inputs & output
  - They will translate as the parameters of your HLS top-level function
- Write up your code
- Test your C++ code
- Synthesis, i.e. convert to VHDL code
  - Optimise it to get the desired performance while staying in your HW limits
- Test synthesised design
- Export design, typically in Vivado IP (Intellectual Property) format
- Implement in Vivado on actual FPGA

# Building and optimising a project

---

- Problem definition:
  - We want to design a high-throughput vector adder and multiplier
    - *Throughput: amount of data items passing through the process*

$$\text{Throughput} = \text{data in input} \frac{\text{clock frequency}}{\text{initiation interval}}$$

- Input & output definition
  - *We receive two 100-dimensional vector of 16-bit signed integer*
  - *We output a 100-dimensional vector of 16-bit signed integer as the sum and an additional 16-bit integer as the product*

Code time!



<https://github.com/simonecid/VivadoTutorial>

- Before optimising your design, you need a reliable system to check that it works as expected
- Testbench!
  - C++ which runs your HLS function with a defined sets of inputs, of which you already know the output
    - e.g. two vectors you know the sum and product of
- Having a test bench that runs through tests is extremely beneficial
  - You can use it to keep on checking that your code keeps on working fine after you have altered it
    - After going through synthesis you might want to redesign parts of it in order to better suit your needs or optimise it
- Typical test runs the function and checks its results via C asserts
  - More extensive and sophisticated test unit libraries, e.g. CPPunit, are available, but let's keep it simple :)



- Add test bench files with
  - **add\_files -tb "FILE"**
- Run your test bench with
  - **csim**
    - Abbreviation of **csim\_design**

```
//checking sum and product of non-null vectors...
for (int a = 0; a < ARRAY_SIZE; a++)
{
    lArray1[a] = 0;
    lArray2[a] = 0;
}

lArray1[10] = 12;
lArray2[10] = 90;
lArray1[67] = 67;
lArray2[67] = -2;
lArray2[88] = -179;

hls_vectorOperations(lArray1, lArray2, lArray3, lProduct);

for (int a = 0; a < ARRAY_SIZE; a++)
{
    switch (a) {
        case 10: assert(lArray3[a] == 102); break;
        case 67: assert(lArray3[a] == 65); break;
        case 88: assert(lArray3[a] == -179); break;
        default: assert(lArray3[a] == 0); break;
    }
}

assert(lProduct == 946);
```

- If the design is working and has been tested, you can proceed with the synthesis
  - Run **csyn** (abbreviation of **csynth\_design**)
- Vivado HLS synthesises VHDL and Verilog (another HDL language) from your C++ code
- Synthesis starts from a top-level function, declared in your .tcl file with **set\_top**
- Parameters of the top-level functions are translated into ports, by default:
  - N-bit variables are translated into STD\_LOGIC\_VECTORS, i.e. array of 1-bit ports
  - Structs and classes are converted to ports by creating ports for each one of their attributes
  - Arrays are translated into ports able to read from an external memory

- After synthesis, HLS produces a report describing the performance of your design under `<ProjectName>/<SolutionName>/syn/report/` in .rpt format, human readable, and .xml, useful for automated analysis

**Utilisation estimates:** breakdown of resource usage.  
*Note: LUTs and FFs are typically overestimated, even by a factor 2*

```
=====
= Performance Estimates
=====
+ Timing (ns):
  * Summary:
  +-----+-----+-----+-----+
  | Clock | Target| Estimated| Uncertainty|
  +-----+-----+-----+-----+
  | ap_clk | 3.33 | 2.643 | 0.42 |
  +-----+-----+-----+-----+

+ Latency (clock cycles):
  * Summary:
  +-----+-----+-----+-----+
  | Latency | Interval | Pipeline|
  | min | max | min | max | Type |
  +-----+-----+-----+-----+
  | 802 | 802 | 802 | 802 | none |
  +-----+-----+-----+-----+

+ Detail:
  * Instance: N/A
  * Loop:
    +-----+-----+-----+-----+
    | Latency | Iteration| Initiation Interval | Trip |
    | min | max | Latency | achieved | target | Count| Pipelined|
    +-----+-----+-----+-----+
    | productLoop | 400 | 400 | 4 | - | - | 100 | no |
    | sumLoop | 400 | 400 | 4 | - | - | 100 | no |
    +-----+-----+-----+-----+
```

**Clock estimate:** gives an initial estimate of whether your design meets the required clock period  
*Note: final clock can only be known after implementation on actual device, sometimes HLS really messes up*

**Latency:** minimum and maximum number of clocks to finish processing, may change if you have variable length loops  
**Initiation Interval (II):** number of clocks before new data can be processed  
**Pipeline:** if the function has been pipelined (more on this soon)

**Loop breakdown:** label your loops to make sure you can see and study its performance.

**Trip count** is the number of iteration of the loop

```
=====
= Utilization Estimates
=====
* Summary:
+-----+-----+-----+-----+
| Name | BRAM_18K| DSP48E| FF | LUT |
+-----+-----+-----+-----+
| DSP | - | 1 | - | - |
| Expression | - | - | 0 | 75 |
| FIFO | - | - | - | - |
| Instance | - | - | - | - |
| Memory | - | - | - | - |
| Multiplexer | - | - | - | 104 |
| Register | - | - | 108 | - |
+-----+-----+-----+-----+
| Total | 0 | 1 | 108 | 179 |
+-----+-----+-----+-----+
| Available SLR | 2160 | 2760 | 663360 | 331680 |
+-----+-----+-----+-----+
| Utilization SLR (%) | 0 | ~0 | ~0 | ~0 |
+-----+-----+-----+-----+
| Available | 4320 | 5520 | 1326720 | 663360 |
+-----+-----+-----+-----+
| Utilization (%) | 0 | ~0 | ~0 | ~0 |
+-----+-----+-----+-----+
```

# Post-synthesis analysis breakdown

```

=====
= Utilization Estimates
=====
* Summary:
-----+-----+-----+-----+-----+
| Name          | BRAM_18K| DSP48E| FF  | LUT  |
-----+-----+-----+-----+-----+
| DSP           |         |      1|    -|    - |
| Expression    |         |      -|    0|   75|
| FIFO         |         |      -|    -|    - |
| Instance      |         |      -|    -|    - |
| Memory        |         |      -|    -|    - |
| Multiplexer   |         |      -|    -|  104|
| Register      |         |      -|    -|   108|
-----+-----+-----+-----+-----+
| Total         |         |      0|    1|  108|   179|
-----+-----+-----+-----+-----+
| Available SLR |        2160| 2760| 663360| 331680|
-----+-----+-----+-----+-----+
| Utilization SLR (%) |         0|   ~0|   ~0|   ~0|
-----+-----+-----+-----+-----+
+ Detail:
-----+-----+-----+-----+-----+
* Instance
N/A | Utilization (%) |         0|   ~0|   ~0|   ~0|
-----+-----+-----+-----+-----+
* DSP48:
-----+-----+-----+-----+-----+
| Instance      | Module          | Expression |
-----+-----+-----+-----+-----+
| hls_vectorOperatibkb_U1 | hls_vectorOperatibkb | i0 * i1 + i2 |
-----+-----+-----+-----+-----+
* Memory:
N/A
-----+-----+-----+-----+-----+
* FIFO:
N/A
-----+-----+-----+-----+-----+
* Expression:
-----+-----+-----+-----+-----+
| Variable Name | Operation | DSP48E| FF  | LUT  | Bitwidth P0| Bitwidth P1|
-----+-----+-----+-----+-----+-----+
| addconv_i_fu_177_p2 | +         |      0|    0|   23|         16|         16|
| x_1_fu_165_p2       | +         |      0|    0|   15|          7|          1|
| x_fu_147_p2         | +         |      0|    0|   15|          7|          1|
| tmp_i4_fu_159_p2   | icmp     |      0|    0|   11|          7|          6|
| tmp_i_fu_141_p2    | icmp     |      0|    0|   11|          7|          6|
-----+-----+-----+-----+-----+
| Total           |           |      0|    0|   75|         44|         30|
-----+-----+-----+-----+-----+

```

```

* Multiplexer:
-----+-----+-----+-----+-----+
| Name          | LUT| Input Size| Bits| Total Bits|
-----+-----+-----+-----+-----+
| agg_result_V_i_reg_98 | 9| 2| 16| 32|
| lap_NS_fsm          | 47| 10| 1| 10|
| inVector1_V_address0 | 15| 3| 7| 21|
| inVector2_V_address0 | 15| 3| 7| 21|
| x_i3_reg_122        | 9| 2| 7| 14|
| x_i_reg_111         | 9| 2| 7| 14|
-----+-----+-----+-----+-----+
| Total           | 104| 22| 45| 112|
-----+-----+-----+-----+-----+
* Register:
-----+-----+-----+-----+-----+
| Name          | FF | LUT| Bits| Const Bits|
-----+-----+-----+-----+-----+
| addconv_i_reg_237 | 16| 0| 16| 0|
| agg_result_V_i_reg_98 | 16| 0| 16| 0|
| lap_CS_fsm          | 9| 0| 9| 0|
| reg_133             | 16| 0| 16| 0|
| reg_137             | 16| 0| 16| 0|
| tmp_1_i_reg_222    | 7| 0| 64| 57|
| x_1_reg_217         | 7| 0| 7| 0|
| x_i3_reg_122        | 7| 0| 7| 0|
| x_i_reg_111         | 7| 0| 7| 0|
| x_reg_194           | 7| 0| 7| 0|
-----+-----+-----+-----+-----+
| Total           | 108| 0| 165| 57|
-----+-----+-----+-----+-----+

```

- You can see how your resources are being used
- 1 DSP used by multiplication
- 75 for the sums
- 108 used for temporary memory

- Base throughput: 1.2 Gb/s
- Let's work on improving this
- Introducing three new concepts:
  - **Pipelining**: enables an iteration of a function or a loop to be executed before the previous one is over
    - Increases throughput w/ minimal resource usage increase
  - **Unrolling**: enables multiple iterations of a for loop to run in parallel, if independent
    - Greatly reduces latency and throughput
    - Can have an impact on resource usage based on loop size
  - **Memory partitioning**: splits array (implemented in BRAM1P/2P or memory port by default) into single registers or ports, enable fast parallel memory access

Base design  
throughput  
1.2 Gb/s



- Let's partition the memories and pipeline the main body of the loops
  - Partition in the body of the function where the variable or the parameter is declared; in main:  
`#pragma HLS array_partition variable=inVector1/2/3`
    - Breaks down the memory interface into single 16-bit ports
  - Put this pragma in loop body to pipeline it; in sumLoop and productLoop:  
`#pragma HLS pipeline`
- Following pipelining and partitioning
  - Latency: 802 → 206
  - II: 802 → 206

```
=====
= Utilization Estimates
=====
* Summary:
+-----+-----+-----+-----+
| Name      | BRAM_18K | DSP48E | FF  | LUT  |
+-----+-----+-----+-----+
| DSP       |          |        |    |     |
| Expression|          |        |    |     |
| FIFO      |          |        |    |     |
| Instance  |          |        |    |     |
| Memory    |          |        |    |     |
| Multiplexer|         |        |    |     |
| Register  |          |        |    |     |
+-----+-----+-----+-----+
| Total     |          |        |    |     |
+-----+-----+-----+-----+
| Available SLR | 2160 | 2760 | 663360 | 331680 |
+-----+-----+-----+-----+
| Utilization SLR (%) | 0 | ~0 | ~0 | ~0 |
+-----+-----+-----+-----+
| Available     | 4320 | 5520 | 1326720 | 663360 |
+-----+-----+-----+-----+
| Utilization (%) | 0 | ~0 | ~0 | ~0 |
+-----+-----+-----+-----+
```



```
=====
= Utilization Estimates
=====
* Summary:
+-----+-----+-----+-----+
| Name      | BRAM_18K | DSP48E | FF  | LUT  |
+-----+-----+-----+-----+
| DSP       |          |        |    |     |
| Expression|          |        |    |     |
| FIFO      |          |        |    |     |
| Instance  |          |        |    |     |
| Memory    |          |        |    |     |
| Multiplexer|         |        |    |     |
| Register  |          |        |    |     |
+-----+-----+-----+-----+
| Total     |          |        |    |     |
+-----+-----+-----+-----+
| Available SLR | 2160 | 2760 | 663360 | 331680 |
+-----+-----+-----+-----+
| Utilization SLR (%) | 0 | ~0 | ~0 | ~0 |
+-----+-----+-----+-----+
| Available     | 4320 | 5520 | 1326720 | 663360 |
+-----+-----+-----+-----+
| Utilization (%) | 0 | ~0 | ~0 | ~0 |
+-----+-----+-----+-----+
```

Base design  
throughput  
1.2 Gb/s

---



Pipelined  
throughput  
4.7 Gb/s

---



- Let's unroll the loops
  - Instead of instantiate logic for a single loop and execute it 100 times, instantiate logic for each iteration and execute in parallel
    - Essentially you increase resource usage by a factor 100
      - DSP: 1  $\rightarrow$  100
  - Put this pragma in loop body to unroll it; in sumLoop and productLoop:  
`#pragma HLS unroll`
- Latency: 206  $\rightarrow$  8
- II: 206  $\rightarrow$  8

```
=====
= Utilization Estimates
=====
* Summary:
+-----+-----+-----+-----+
| Name | BRAM_18K | DSP48E | FF | LUT |
+-----+-----+-----+-----+
| DSP | | 1 | | |
| Expression | | | 0 | 30 |
| FIFO | | | | |
| Instance | | | 10 | 2099 |
| Memory | | | | |
| Multiplexer | | | | 63 |
| Register | | | 65 | |
+-----+-----+-----+-----+
| Total | 0 | 1 | 75 | 2192 |
+-----+-----+-----+-----+
| Available SLR | 2160 | 2760 | 663360 | 331680 |
+-----+-----+-----+-----+
| Utilization SLR (%) | 0 | ~0 | ~0 | ~0 |
+-----+-----+-----+-----+
| Available | 4320 | 5520 | 1326720 | 663360 |
+-----+-----+-----+-----+
| Utilization (%) | 0 | ~0 | ~0 | ~0 |
+-----+-----+-----+-----+
```



```
=====
= Utilization Estimates
=====
* Summary:
+-----+-----+-----+-----+
| Name | BRAM_18K | DSP48E | FF | LUT |
+-----+-----+-----+-----+
| DSP | | | | | |
| Expression | | | | |
| FIFO | | | | |
| Instance | | | 100 | 1736 | 2953 |
| Memory | | | | |
| Multiplexer | | | | 15 |
| Register | | | | 3 |
+-----+-----+-----+-----+
| Total | 0 | 100 | 1739 | 2968 |
+-----+-----+-----+-----+
| Available SLR | 2160 | 2760 | 663360 | 331680 |
+-----+-----+-----+-----+
| Utilization SLR (%) | 0 | 3 | ~0 | ~0 |
+-----+-----+-----+-----+
| Available | 4320 | 5520 | 1326720 | 663360 |
+-----+-----+-----+-----+
| Utilization (%) | 0 | 1 | ~0 | ~0 |
+-----+-----+-----+-----+
```



Base design  
throughput  
1.2 Gb/s

---



Pipelined  
throughput  
4.7 Gb/s

---



Unrolled  
throughput  
120 Gb/s



# Pipelining the top-level function

- The pipeline pragma pipelines the function in which it is located and unroll and pipelines every underlying loop
  - If we place a pipeline pragma in the top-level function body, everything will be unrolled and pipelined, maximising performance
- Latency: 8  $\rightarrow$  8
- II: 8  $\rightarrow$  1, data can be input every clock cycle, **max. throughput**

```
=====
= Utilization Estimates
=====
* Summary:
+-----+-----+-----+-----+
| Name | BRAM_18K | DSP48E | FF | LUT |
+-----+-----+-----+-----+
| DSP | | | | |
| Expression | | | | |
| FIFO | | | | |
| Instance | | 100 | 1736 | 2953 |
| Memory | | | | |
| Multiplexer | | | | 15 |
| Register | | | | 3 |
+-----+-----+-----+-----+
| Total | 0 | 100 | 1739 | 2968 |
+-----+-----+-----+-----+
| Available SLR | 2160 | 2760 | 663360 | 331680 |
+-----+-----+-----+-----+
| Utilization SLR (%) | 0 | 3 | ~0 | ~0 |
+-----+-----+-----+-----+
| Available | 4320 | 5520 | 1326720 | 663360 |
+-----+-----+-----+-----+
| Utilization (%) | 0 | 1 | ~0 | ~0 |
+-----+-----+-----+-----+
```



```
=====
= Utilization Estimates
=====
* Summary:
+-----+-----+-----+-----+
| Name | BRAM_18K | DSP48E | FF | LUT |
+-----+-----+-----+-----+
| DSP | | | | | |
| Expression | | | | 0 | 4 |
| FIFO | | | | |
| Instance | | 0 | 100 | 7217 | 3942 |
| Memory | | | | |
| Multiplexer | | | | |
| Register | | 0 | | 1609 | 800 |
+-----+-----+-----+-----+
| Total | 0 | 100 | 8826 | 4746 |
+-----+-----+-----+-----+
| Available SLR | 2160 | 2760 | 663360 | 331680 |
+-----+-----+-----+-----+
| Utilization SLR (%) | 0 | 3 | 1 | 1 |
+-----+-----+-----+-----+
| Available | 4320 | 5520 | 1326720 | 663360 |
+-----+-----+-----+-----+
| Utilization (%) | 0 | 1 | ~0 | ~0 |
+-----+-----+-----+-----+
```

# Pipelining the top-level function

Base design  
throughput  
1.2 Gb/s

---



Pipelined  
throughput  
4.7 Gb/s

---



Unrolled  
throughput  
120 Gb/s

---



Fully  
pipelined  
throughput  
960 Gb/s



- Whenever you create a function, HLS creates a separate logic block and connects it to the logic block of the main function
  - Increases latency
  - Prevents HLS from running optimisations that reduces resource usage
  - In the function body (not top-level): `#pragma HLS inline`
    - Inlines and integrates the sub-function in the calling one
- Latency: 8 → 7


```
=====
= Utilization Estimates
=====
* Summary:
+-----+-----+-----+-----+
| Name      | BRAM_18K | DSP48E | FF  | LUT |
+-----+-----+-----+-----+
| DSP       |          |        |     |     |
| Expression|          |        | 0   | 4   |
| FIFO      |          |        |     |     |
| Instance  | 0        | 100    | 7217| 3942|
| Memory    |          |        |     |     |
| Multiplexer|         |        |     |     |
| Register  | 0        |        | 1609| 800 |
+-----+-----+-----+-----+
| Total     | 0        | 100    | 8826| 4746|
+-----+-----+-----+-----+
| Available SLR | 2160    | 2760   | 663360| 331680|
+-----+-----+-----+-----+
| Utilization SLR (%) | 0       | 3      | 1     | 1   |
+-----+-----+-----+-----+
| Available    | 4320    | 5520   | 1326720| 663360|
+-----+-----+-----+-----+
| Utilization (%) | 0       | 1      | ~0    | ~0  |
+-----+-----+-----+-----+
```



```
=====
= Utilization Estimates
=====
* Summary:
+-----+-----+-----+-----+
| Name      | BRAM_18K | DSP48E | FF  | LUT |
+-----+-----+-----+-----+
| DSP       |          |        |     |     |
| Expression|          |        | 0   | 2913|
| FIFO      |          |        |     |     |
| Instance  |          |        |     |     |
| Memory    |          |        |     |     |
| Multiplexer|         |        |     |     |
| Register  | 0        |        | 5608| 1824|
+-----+-----+-----+-----+
| Total     | 0        | 100    | 5608 | 4737|
+-----+-----+-----+-----+
| Available SLR | 2160    | 2760   | 663360| 331680|
+-----+-----+-----+-----+
| Utilization SLR (%) | 0       | 3      | ~0    | 1   |
+-----+-----+-----+-----+
| Available    | 4320    | 5520   | 1326720| 663360|
+-----+-----+-----+-----+
| Utilization (%) | 0       | 1      | ~0    | ~0  |
+-----+-----+-----+-----+
```



- Synthesised design can be tested in HDL simulator in the C test bench
  - Run **cosim** (abbreviation of **cosim\_design**)
    - First tests the C code, then the synthesised design
- If everything looks good, you can export it for actual implementation
  - Using IP catalog now, but other formats are available
  - Final product of Vivado HLS
  - **export\_design -format ip\_catalog**
    - Exported design can be found in `<ProjectName>/<SolutionName>/impl/ip`
- From here on is Vivado domain, not covered here, but you can load IP and implement it



```
inVector1_90_V[15:0]
inVector1_91_V[15:0]
inVector1_92_V[15:0]
inVector1_93_V[15:0]
inVector1_94_V[15:0]
inVector1_95_V[15:0]
inVector1_96_V[15:0]
inVector1_97_V[15:0]
inVector1_98_V[15:0]
inVector1_99_V[15:0]
inVector2_0_V[15:0]
inVector2_1_V[15:0]
inVector2_2_V[15:0]
inVector2_3_V[15:0]
inVector2_4_V[15:0]
inVector2_5_V[15:0]
inVector2_6_V[15:0]

outVectorSum_93_V_ap_vld
outVectorSum_94_V_ap_vld
outVectorSum_95_V_ap_vld
outVectorSum_96_V_ap_vld
outVectorSum_97_V_ap_vld
outVectorSum_98_V_ap_vld
outVectorSum_99_V_ap_vld
outProduct_V_ap_vld
outVectorSum_0_V[15:0]
outVectorSum_1_V[15:0]
outVectorSum_2_V[15:0]
outVectorSum_3_V[15:0]
outVectorSum_4_V[15:0]
outVectorSum_5_V[15:0]
outVectorSum_6_V[15:0]
outVectorSum_7_V[15:0]
outVectorSum_8_V[15:0]
```

# Tips and tricks

---

- You can use C++11 and higher constructs, e.g. auto or constexpr:  
`add_files -cflags "-std=c++11 "<HLS_FILE>"`
- Run thorough tests on software, do not be lazy like me!
  - Debugging stuff at later stages is just way harder and confusing
    - If you do not trust me, ask Aaron!
- Read the [list of pragmas](#) and experiment a lot with them
  - Array\_partition, pipeline, and unroll accept options, study them!
  - Pragmas try to bridge the gap between C++ and HLS, master them
- HLS likes ternary operators, if possible use them instead of if statements!

Ternary operator → `a = (b > 0) ? c : d;`

Equivalent if statement

```
if (b > 0)
  a = c;
else
  a = d;
```

# Various tips and tricks

## *Splitting designs*

- Big designs take long to synthesise
- Split your problem in smaller projects
- Each project can be exported in IP format and then linked in a chain
- Saves lots of synthesis time
- Increases flexibility
  - Blocks can be run at different clock speeds
- Example: the jet trigger algorithm I work on is made of three blocks
  - Histogrammer
  - Data buffer
  - Jet finder
- **Divide et impera reigns!**



History taught us that this strategy works!



- Your time is precious!  
Do not waste it implementing large broken designs.
- Start small and write code that can be easily scaled up!
- For instance, let's say you need to do some processing on a large number of inputs
  - Make the number of inputs a parameter of your code with a `#define NUMBER_OF_INPUTS XX` and make your code depend on it
  - Do your initial testing on a scaled-down version of your code, i.e. with few inputs, then increase it
    - Takes way less time to implement a smaller design

- Final timing and resource usage results are only obtainable after implementation
- Vivado HLS provides tools to implement design without using Vivado
  - Not sure how it works, I presume it makes some basic assumptions on how you are going to place your design in a FPGA and implements it
- By running it you can get a more accurate estimates of timing and resource usage, although not final they tend to be much closer
- Run **export\_design -format ip\_catalog -evaluate vhdl**
  - This implements the VHDL design on FPGA
  - 10 minutes to run for the small test design, against XX for synthesis
  - Results in *<ProjectName>/<SolutionName>/impl/report/vhdl/*

# Various tips and tricks

## Getting more accurate estimates

```
#=== Post-Implementation Resource usage ===
CLB: ..... 1007
LUT: ..... 2897 # to be added to SRL 3697 vs 4737
FF: ..... 8551 # was 5608
DSP: ..... 100 # was 100
BRAM: ..... 0
SRL: ..... 800 # to be added to LUT 3697 vs 4737
#=== Final timing ===
CP required: ... 3.333
CP achieved post-synthesis: ... 2.302
CP achieved post-implementation: ... 3.228 # was 2.693+-0.42
Timing met
```

- FOR THE LOVE OF GOD DO NOT USE THE C/C++ STANDARD LIBRARY!
  - I have heard it gives horrible results
    - I do not even know how they managed to get HLS to synthesise
- Do not reinvent the wheel!
  - Vivado HLS has libraries doing many interesting things
    - It is all in the manual
  - For instance, `#include <hls_math.h>` for HLS math libraries

# Using C++ constructs

---

Code time!



[https://github.com/simonecid/VivadoTutorial/tree/cpp\\_version](https://github.com/simonecid/VivadoTutorial/tree/cpp_version)

- Rewritten the vector add and multiply by developing a generic Vector class via template
  - Generic, flexible, easy to use
    - N-dimensional
    - Uses any type
- Same resource usage
- Clever usage of C++ constructs provides great flexibility without usage penalties
- Note:
  - Partitioning of class attributes must be invoked in constructors
  - Inline every class method!

```
+ Latency (clock cycles):
* Summary:
+-----+-----+-----+-----+
| Latency | Interval | Pipeline |
| min | max | min | max | Type |
+-----+-----+-----+-----+
| 7 | 7 | 1 | 1 | function |
+-----+-----+-----+-----+
```

```
=====
= Utilization Estimates
=====
* Summary:
+-----+-----+-----+-----+
| Name | BRAM_18K | DSP48E | FF | LUT |
+-----+-----+-----+-----+
| DSP | - | 100 | - | - |
| Expression | - | - | 0 | 2913 |
| FIFO | - | - | - | - |
| Instance | - | - | - | - |
| Memory | - | - | - | - |
| Multiplexer | - | - | - | - |
| Register | 0 | - | 5608 | 1824 |
+-----+-----+-----+-----+
| Total | 0 | 100 | 5608 | 4737 |
+-----+-----+-----+-----+
| Available SLR | 2160 | 2760 | 663360 | 331680 |
+-----+-----+-----+-----+
| Utilization SLR (%) | 0 | 3 | ~0 | 1 |
+-----+-----+-----+-----+
| Available | 4320 | 5520 | 1326720 | 663360 |
+-----+-----+-----+-----+
| Utilization (%) | 0 | 1 | ~0 | ~0 |
+-----+-----+-----+-----+
```





- HLS enables users to write FPGA firmware in high-level languages
  - More flexible and easier to use
- HLS pragmas can be used to produce high-throughput designs
  - Pipeline functions, unroll loops and partition memory
  - Used it on a vector adder and multiplier
- The machine extension is available in Bristol per FPGA development
- Went through a number of tips and tricks
- Using C++ classes and template does not affect resource usage while improving code flexibility and ease of use
- Collection of my FPGA bookmarks in next slide
- Contacts:
  - [simone.bologna@bristol.ac.uk](mailto:simone.bologna@bristol.ac.uk)
  - Skype: simonecid
  - Office: 4.57

- HLS guide by Xilinx,  
[https://www.xilinx.com/support/documentation/sw\\_manuels/xilinx2018\\_2/ug902-vivado-high-level-synthesis.pdf](https://www.xilinx.com/support/documentation/sw_manuels/xilinx2018_2/ug902-vivado-high-level-synthesis.pdf)
- Optimisation in HLS by Xilinx  
[https://www.xilinx.com/support/documentation/sw\\_manuels/xilinx2018\\_1/ug1270-vivado-hls-opt-methodology-guide.pdf](https://www.xilinx.com/support/documentation/sw_manuels/xilinx2018_1/ug1270-vivado-hls-opt-methodology-guide.pdf)
- Pipelining and Unrolling tips,  
[https://www.xilinx.com/support/documentation/sw\\_manuels/xilinx2015\\_2/sdsoc\\_doc/topics/calling-coding-guidelines/concept\\_pipelining\\_loop\\_unrolling.html](https://www.xilinx.com/support/documentation/sw_manuels/xilinx2015_2/sdsoc_doc/topics/calling-coding-guidelines/concept_pipelining_loop_unrolling.html)
- Parallelising function tip,  
<https://forums.xilinx.com/t5/Vivado-High-Level-Synthesis-HLS/How-to-set-the-two>
- HLS tips, <https://fling.seas.upenn.edu/~giesen/dynamic/wordpress/vivado-hls-learnings/>
- HLS pragma list,  
[https://www.xilinx.com/html\\_docs/xilinx2018\\_3/sdsoc\\_doc/hls-pragmas-okr1504034364623.html](https://www.xilinx.com/html_docs/xilinx2018_3/sdsoc_doc/hls-pragmas-okr1504034364623.html)
- Introductory slides to HLS,  
[http://home.mit.bme.hu/~szanto/education/vimima15/heterogen\\_vivado\\_hls.pdf](http://home.mit.bme.hu/~szanto/education/vimima15/heterogen_vivado_hls.pdf)
- Improving performance in HLS,  
[http://users.ece.utexas.edu/~gerstl/ee382v\\_f14/soc/vivado\\_hls/VivadoHLS\\_Improving\\_Performance.pdf](http://users.ece.utexas.edu/~gerstl/ee382v_f14/soc/vivado_hls/VivadoHLS_Improving_Performance.pdf)