Introduction to FPGAs

Christopher Brown
Epigraphy Kickoff meeting
12/01/26

What is an FPGA?

- FPGA – Field Programmable Gate Array
  - ASIC that can be updated
  - Means very fast processing
  - FPGAs are not CPUs or GPUs

- CPU programming: asking your friend to make you a cake
- FPGA programming: building a bakery to constantly make cakes

The CPU bakery

Process at one step at a time

Can increase cake throughput by having multiple friends

You can ask them to change the recipe very easily

Your friend can also do lots of other things apart from bake cakes

The FPGA bakery

Process multiple steps at a time

Throughput is limited by the resources, can only put one cake in the oven but can do other things while it bakes

Harder to change the recipe

Would have to reconfigure the bakery to start making pies

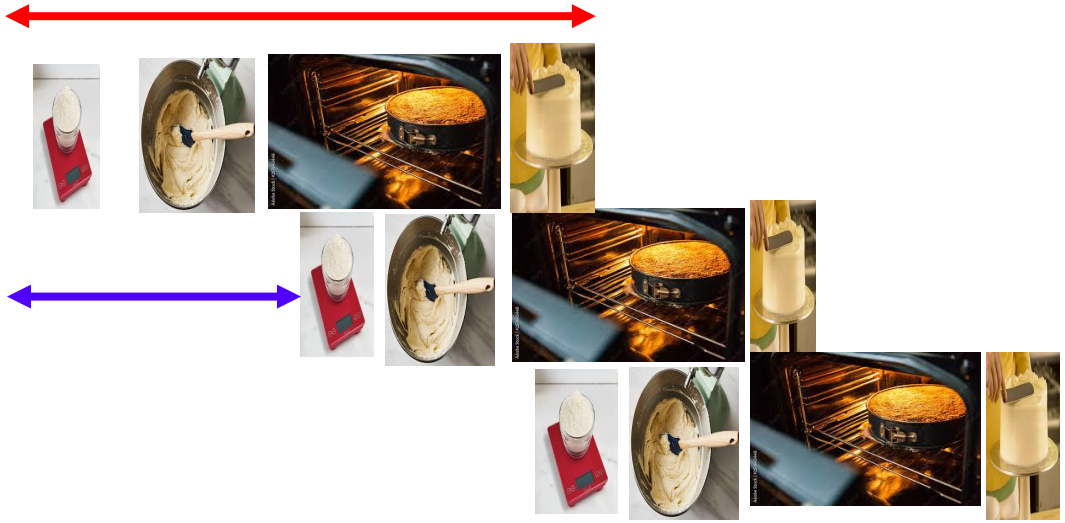
The FPGA bakery

Latency: how long does it take to make a cake

The FPGA bakery

Initiation interval: how long does it take before I can make a cake again
In completely sequential processing, II = Latency
In pipelined processing II < latency, I can be making multiple cakes at once
In this case our II is 1, we are fully pipelined

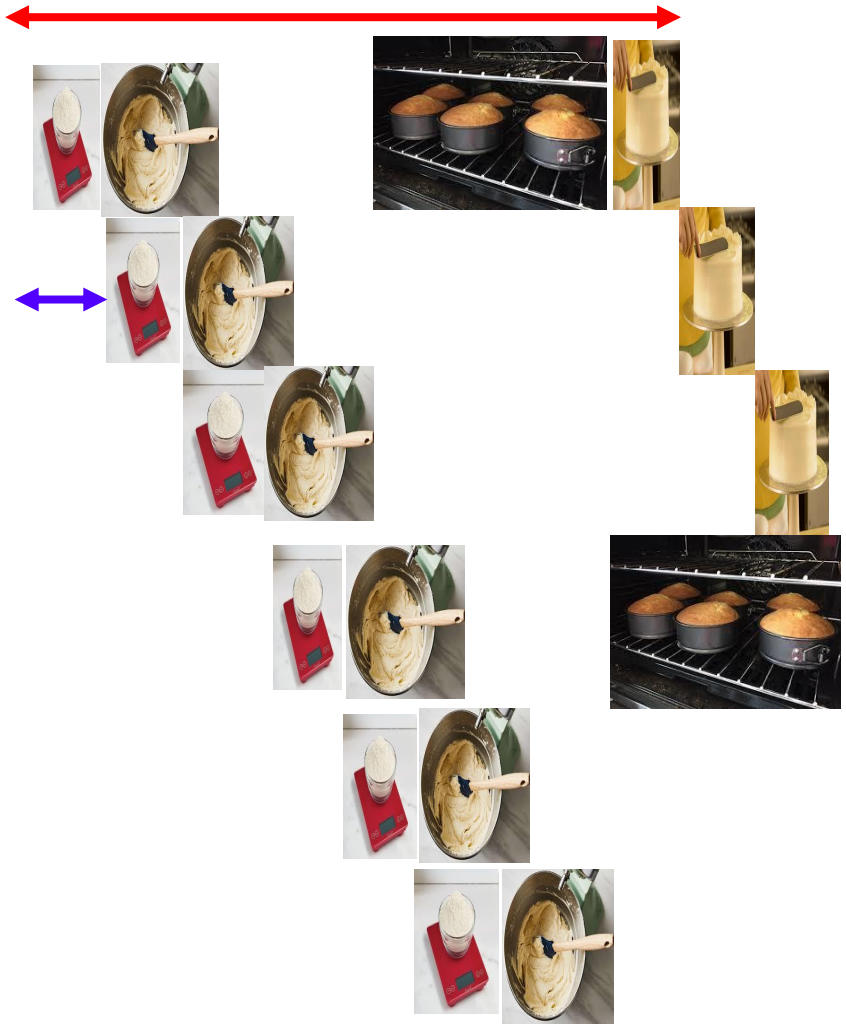
The FPGA bakery



In reality things look a bit different

II for a full processing chain is dictated by the II of the slowest part, in this case the oven

The FPGA bakery



If we design our bakery a bit better we can decrease the II of our process

This needs us to buffer outputs from one process before feeding it to the next

But this increase the latency

The FPGA bakery

If our bakery was very big we could have 5 ovens all running at once to achieve the best possible throughput

Low II and lowest possible latency

Requires expensive ovens and a big floorplan

What did we learn

If you want high cake throughput asking your CPU friend to do it is not a good idea

Using the FPGA bakery will increase your throughput

But it requires careful planning

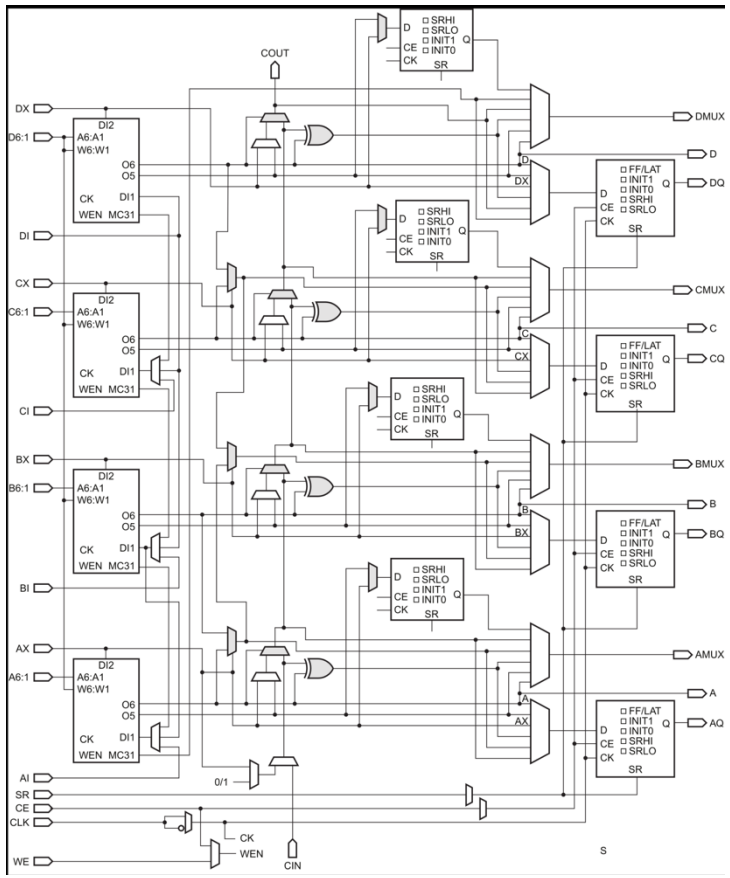
- Rarely do all your processes take the same time
- You need to balance II and latency depending on your cake needs
- You need to think about how cakes flow through the bakery and identify critical paths
- You need to balance your resources, you can't have infinite ovens

It is hard to configure your bakery, it is easy to ask your CPU friend to bake a cake.

In FPGA programming we are not just designing the cake baking steps we are designing the bakery

FPGA building blocks

- Reconfigurable logic gates (don’t need to worry too much about these)



FPGA building blocks

- Look-up tables LUT, inputs determine outputs
  - Can put any function you like
  - Need to worry about size of input and output best for small bitwidth inputs 2-6 bits

FPGA building blocks

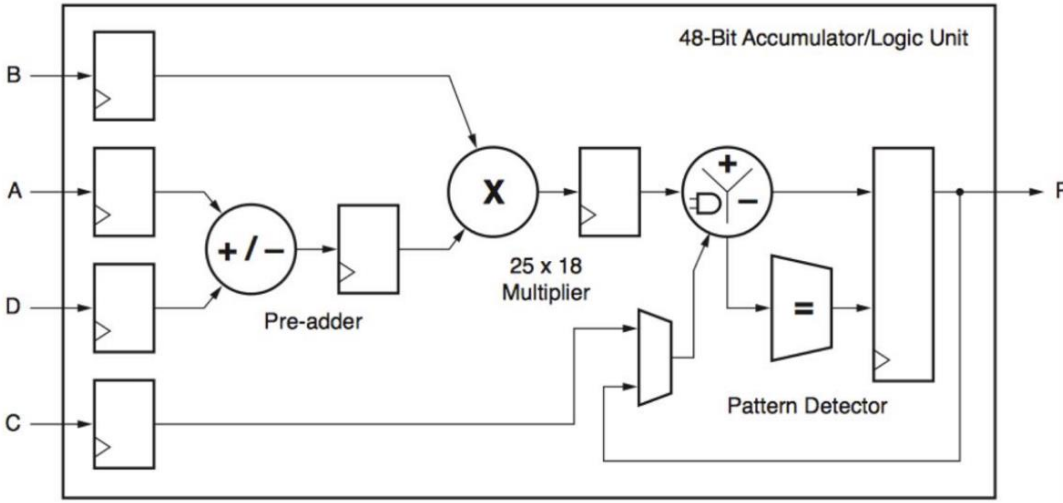
- Flip Flops, keeps the state of a signal between two clock cycles, needed for sequential logic where our system has a state
- Allows us to create registers and pipelines that store signals
- Any kind of digital electronics can be written out of just LUTs and FFs

FPGA building blocks

- Larger addressable memory: Block Random Access Memory, BRAM

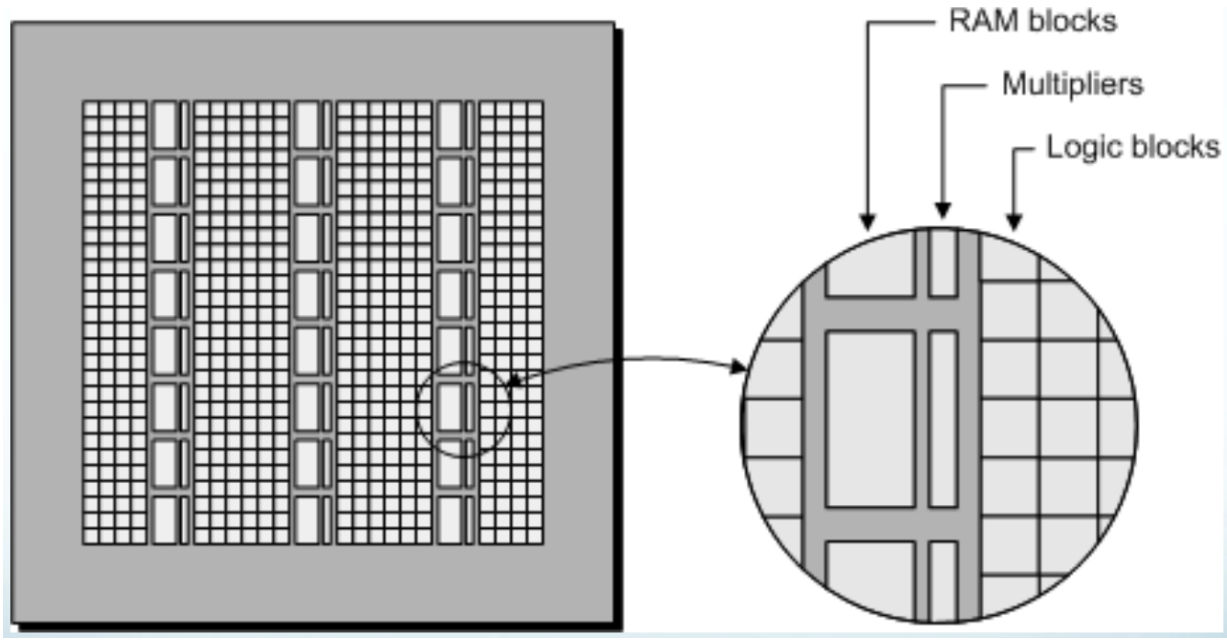
- Dedicated processing for common operations, digital signal processors (DSPs), mostly multiplications

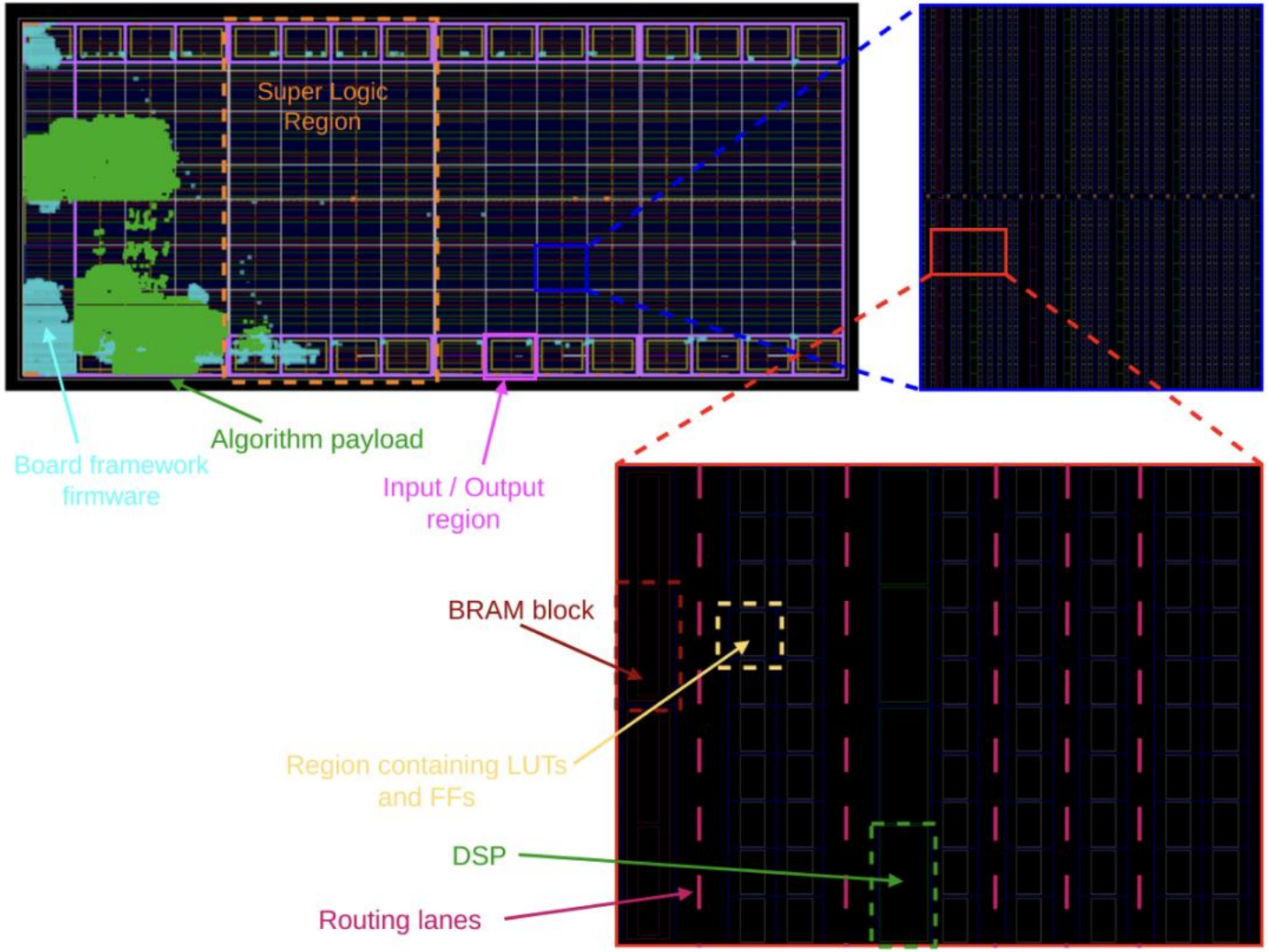
- These are more complicated dedicated circuits, fewer of them
  - Vector multiplication will use DSPs, we want to reduce this if possible
  - Small input width multiplication will be implemented as a LUT



FPGA fabric

- FPGAs are arranged in columns: logic blocks consisting of LUTs and FFs, RAM blocks, and Multipliers
- Network of programmable routing logic
- Clock tree to drive signals



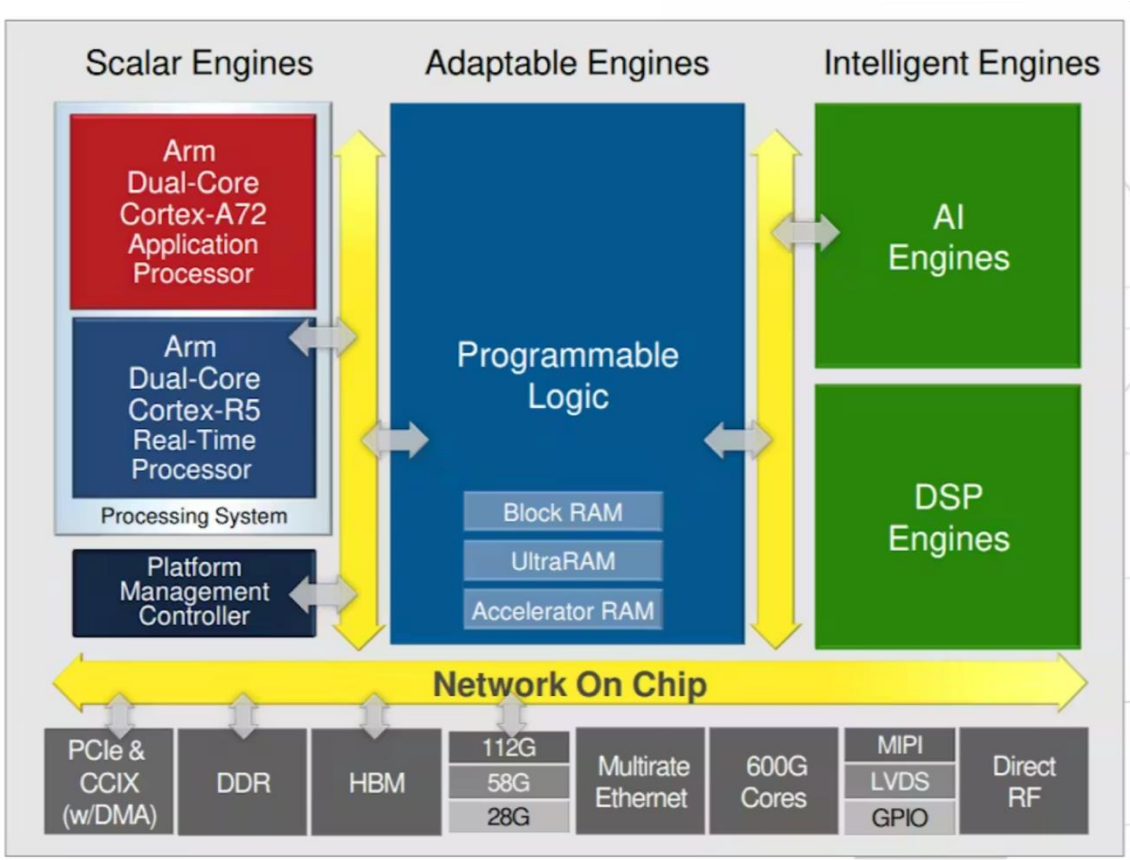


- Big FPGAs are divided into “super logic regions”
- In reality these SLRs are stacked silicon with high-speed interconnects between them

CMS Era	Years	FPGA	Logic Cells	BRAM (Mb)	DSPs
Original	2008–2016	Virtex-II	32,000	1.7	96
Phase-1	2016–2025	Virtex 7	693,000	52.9	3,600
Phase-2	2028–Onwards	Virtex Ultrascale+ VU13P	3,780,000	455.0	12,288

- Approximate resources available on high end FPGAs in a given era

New trends



- Xilinx Versal chip

FPGA programming

- Can’t just send instructions to the FPGA like a CPU
- Generally, don’t need to specifically program logic gates
- Two main methods:
  - Hardware description language HDL -> VHDL or Verilog
  - High-level synthesis -> Looks like C++ but with extras

FPGA programming

- HDL: designing the bakery yourself: need a high level of understanding of the bakery components but can hopefully solve problems
- HLS: asking a bakery designer to design it for you: they might do something a bit weird but it'll work. If something goes wrong you won't understand what they did

HDL - example

```
entity add is
    port(
        i_clk : IN std_logic := '0';
        i_A : IN std_logic_vector( 31 downto 0);
        i_B : IN std_logic_vector( 31 downto 0);
        o_C : OUT std_logic_vector( 31 downto 0)
    );
    end add;

architecture rtl of add is
    if rising_edge(i_clk) then
        o_C <= i_A + i_B;
    end if;
end rtl;
```

Won't go into detail as it is a whole language in itself

Whatever A and B are at a given clock cycle they will be added together and C will be the output

This will happen at every clock cycle no matter what

A and B are signals within the circuit that will be plugged into other signals that ultimately come from your board input

Overflow can and will happen if A and B close to 2^32

FPGA programming

- High level synthesis
- Tool that translates C++ to HDL
- Compiler decides how to implement your code as HDL
  - You have some control over it, can specify how you want the processing to happen using pragmas, fully parallelise, targeting a specific latency
  - HLS will produce a set of HDL files that you can integrate in the FPGA, this HDL is mostly meaningless and debugging on the FPGA hard
  - The resources used are similar to a direct HDL approach, and sometimes better

  - Will show vitis hls for AMD FPGAs, other hls's are available depending on your FPGA

HLS example

```
int add( int i_A, int i_B){
    return i_A + i_B;
}
```

HLS example

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity add is
port (
    ap_start : IN STD_LOGIC;
    ap_done : OUT STD_LOGIC;
    ap_idle : OUT STD_LOGIC;
    ap_ready : OUT STD_LOGIC;
    i_a : IN STD_LOGIC_VECTOR (31 downto 0);
    i_b : IN STD_LOGIC_VECTOR (31 downto 0);
    ap_return : OUT STD_LOGIC_VECTOR (31 downto 0);
    ap_rst : IN STD_LOGIC );
end;


architecture behav of add is
    attribute CORE_GENERATION_INFO : STRING;
    attribute CORE_GENERATION_INFO of behav : architecture is
    "add_add,hls_ip_2024_1,{HLS_INPUT_TYPE=cxx,HLS_INPUT_FLOAT=0,HLS_INPUT_FIXED=0,HLS_INPUT_PART=xcvu9p-flga2104-2-i,HLS_INPUT_CLOCK=5.000000,HLS_INPUT_ARCH=others,HLS_S
    constant ap_const_logic_1 : STD_LOGIC := '1';
    constant ap_const_logic_0 : STD_LOGIC := '0';
    constant ap_const_boolean_1 : BOOLEAN := true;

    signal ap_ce_reg : STD_LOGIC;


begin



    ap_done <= ap_start;
    ap_idle <= ap_const_logic_1;
    ap_ready <= ap_start;
    ap_return <= std_logic_vector(unsigned(i_b) + unsigned(i_a));
end behav;
```

Output HDL from the HLS

Would need plugging into a wider HDL project that defines the various input and output signals

These get very unwieldy as the function complexity increases

FPGA design flow

- Work out what you want your algorithm to do
  - How does data flow through your algorithm
  - What are the key computations you need to do

FPGA design flow

- What do your inputs look like? All inputs to an algorithm may not be available at the same time
  - Are my inputs ordered and sorted?
  - Do I need to do pairwise comparison?
  - Do I need to do complex preprocessing for my algorithm?

FPGA design flow

- What do your outputs look like?
  - Do I also need to order these?
  - Do I need that sigmoid activation function?

FPGA design flow

- What are my constraints
  - What is the FPGA I'm targeting
  - What is the clock frequency I'm targeting
  - Does it need to be fully pipelined II=1
  - Do other algorithms need to be on this FPGA?

```
set project_name "project_test"
set backend "vivado"
set part "xcvu13p-flga2577-2-e"
set clock_period 2.5
set clock_uncertainty 27%
```

FPGA design flow

- Write our algorithm using HLS (so that later we can see these things in practice with hls4ml)
- Also write a testbench that simulates the data being passed through the algorithm
- We will discuss high level concepts, not low-level coding

FPGA design flow

- Run a C-simulation that compiles our algorithm code and then runs it through our testbench
  - This will verify the output of the algorithm is doing what we expect, either we can compare to some predetermined output or we can use C++ as the comparison
  - This is essential to correctly test the functionality your hls code


```
-5.4082 0
INFO: Saved inference results to file: tb_data/csim_results.log
INFO: [SIM 1] CSim done with 0 errors.
INFO: [SIM 3] *************** CSIM finish ***************
```

- Can use print statements in your testbench to verify your code

FPGA design flow

- Now we can run a c-synthesis which translates your HLS to a register transfer level (RTL)

FPGA design flow

- C-Synths will spit out long logs
  - It is sometimes worth checking through these -> vitis_hls.log
  - Search for “WARNING” or “ERROR”

```
****** Vitis HLS - High-Level Synthesis from C, C++ and OpenCL v2023.2 (64-bit)
  **** SW Build 4023990 on Oct 11 2023
  **** IP Build 4028589 on Sat Oct 14 00:45:43 MDT 2023
  **** SharedData Build 4025554 on Tue Oct 10 17:18:54 MDT 2023
    ** Copyright 1986-2022 Xilinx, Inc. All Rights Reserved.
    ** Copyright 2022-2023 Advanced Micro Devices, Inc. All Rights Reserved.

source /opt/Xilinx/Vitis_HLS/2023.2/scripts/vitis_hls/hls.tcl -notrace
INFO: [HLS 200-10] Running '/opt/Xilinx/Vitis_HLS/2023.2/bin/unwrapped/lnx64.o/vitis_hls'
INFO: [HLS 200-10] For user 'root' on host '00815598c4d5' (Linux_x86_64 version 5.14.0-611.9.1.el9_7.x86_64) on Thu Jan 08 12:08
INFO: [HLS 200-10] In directory '/root/TrainTagger/output/baseline/firmware/L1TSC4NGJetModel'
INFO: [HLS 200-2053] The vitis_hls executable is being deprecated. Consider using vitis-run --mode hls --tcl
Sourcing Tcl script 'build_prj.tcl'
INFO: [HLS 200-1510] Running: open_project L1TSC4NGJetModel_prj
INFO: [HLS 200-10] Opening project '/root/TrainTagger/output/baseline/firmware/L1TSC4NGJetModel/L1TSC4NGJetModel_prj'.
INFO: [HLS 200-1510] Running: set_top L1TSC4NGJetModel
INFO: [HLS 200-1510] Running: add_files firmware/L1TSC4NGJetModel.cpp -cflags -std=c++0x
INFO: [HLS 200-10] Adding design file 'firmware/L1TSC4NGJetModel.cpp' to the project
INFO: [HLS 200-1510] Running: add_files -tb L1TSC4NGJetModel_test.cpp -cflags -std=c++0x
INFO: [HLS 200-10] Adding test bench file 'L1TSC4NGJetModel_test.cpp' to the project
INFO: [HLS 200-1510] Running: add_files -tb firmware/weights
INFO: [HLS 200-10] Adding test bench file 'firmware/weights' to the project
INFO: [HLS 200-1510] Running: add_files -tb tb_data
INFO: [HLS 200-10] Adding test bench file 'tb_data' to the project
INFO: [HLS 200-1510] Running: open_solution solution1
INFO: [HLS 200-10] Opening solution '/root/TrainTagger/output/baseline/firmware/L1TSC4NGJetModel/L1TSC4NGJetModel_prj/solution1'.
```

FPGA design flow

- You will also get an estimation of the maximum clock speed of your design, search for “Fmax” in the vitis_hls.log

```
INFO: [VLOG 209-307] Generating Verilog RTL for my_prj.
INFO: [HLS 200-789] **** Estimated Fmax: 312.71 MHz
```

- Don’t assume this means your design will actually run at this clock speed….

- csynth latency estimates are normally pretty accurate as is the initiation interval
- Resource utilization estimates are not good, typically LUT usage is far higher and DSP usage is lower than in reality
- Don't believe these estimates, and don't believe papers than only quote csynth estimates

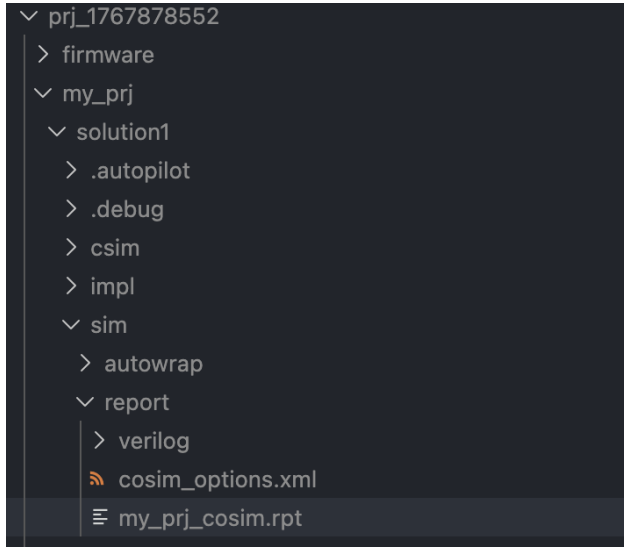

```
+ Timing:
    * Summary:
    +--------+----------+----------+------------+
    |  Clock |  Target  | Estimated| Uncertainty|
    +--------+----------+----------+------------+
    |ap_clk  |  5.00 ns|  3.198 ns|     1.35 ns|
    +--------+----------+----------+------------+

+ Latency:
    * Summary:
    +---------+---------+-----------+-----------+-----+-----+---------+
    |  Latency (cycles) |   Latency (absolute)  |  Interval | Pipeline|
    |   min   |   max   |    min    |    max    | min | max |   Type  |
    +---------+---------+-----------+-----------+-----+-----+---------+
    |        2|        2|  10.000 ns|  10.000 ns|    1|    1|      yes|
    +---------+---------+-----------+-----------+-----+-----+---------+
```

```
================================================================
== Utilization Estimates
================================================================
* Summary:
+-----------------------+---------+-----+---------+---------+-----+
|          Name         | BRAM_18K|  DSP |    FF   |   LUT   | URAM|
+-----------------------+---------+-----+---------+---------+-----+
|DSP                    |        -|    -|        -|        -|    -|
|Expression             |        -|    -|        0|        2|    -|
|FIFO                   |        -|    -|        -|        -|    -|
|Instance               |        -|    -|     2181|    64588|    -|
|Memory                 |        -|    -|        -|        -|    -|
|Multiplexer            |        -|    -|        -|        -|    -|
|Register               |        -|    -|        3|        -|    -|
+-----------------------+---------+-----+---------+---------+-----+
|Total                  |        0|    0|     2184|    64590|    0|
+-----------------------+---------+-----+---------+---------+-----+
|Available SLR          |     1440|  2280|   788160|   394080|  320|
+-----------------------+---------+-----+---------+---------+-----+
|Utilization SLR (%)    |        0|    0|       ~0|       16|    0|
+-----------------------+---------+-----+---------+---------+-----+
|Available              |     4320|  6840|  2364480|  1182240|  960|
+-----------------------+---------+-----+---------+---------+-----+
|Utilization (%)        |        0|    0|       ~0|        5|    0|
+-----------------------+---------+-----+---------+---------+-----+
```

FPGA design flow

- We can now also do a “Co-simulation” that uses the synthesised code and simulates the HDL
- This will give you an accurate estimate of the latency and initiation interval



RTL	Status	Latency(Clock Cycles)			Interval(Clock Cycles)			Total Execution Time (Clock Cycles)
		min	avg	max	min	avg	max	
VHDL	NA	NA	NA	NA	NA	NA	NA	NA
Verilog	Pass	2	2	2	NA	NA	NA	2

FPGA design flow

- This is typically where everyone stops
- Don’t assume a “synthesisable” bit of code actually runs in hardware
- There is still a lot more work implementing an algorithm

FPGA design flow

- We can also do a “vsynth” a vivado synth which maps the RTL to logic gates or “netlist”
- The resource estimates of the vivado synth are typically a lot better
- They are still “out-of-context” meaning they are standalone without surrounding FW

- In the vivado_synth.rpt

Site Type	Used	Fixed	Prohibited	Available	Util%
CLB LUTs*	7866	0	0	1182240	0.67
LUT as Logic	7866	0	0	1182240	0.67
LUT as Memory	0	0	0	591840	0.00
CLB Registers	1521	0	0	2364480	0.06
Register as Flip Flop	1521	0	0	2364480	0.06
Register as Latch	0	0	0	2364480	0.00
CARRY8	1294	0	0	147780	0.88
F7 Muxes	1	0	0	591120	<0.01
F8 Muxes	0	0	0	295560	0.00
F9 Muxes	0	0	0	147780	0.00

2. BLOCKRAM

Site Type	Used	Fixed	Prohibited	Available	Util%
Block RAM Tile	0	0	0	2160	0.00
RAMB36/FIFO*	0	0	0	2160	0.00
RAMB18	0	0	0	4320	0.00
URAM	0	0	0	960	0.00

* Note: Each Block RAM Tile only has one FIFO logic available and

3. ARITHMETIC

Site Type	Used	Fixed	Prohibited	Available	Util%
DSPs	0	0	0	6840	0.00

Vsynth

Site Type	Used	Fixed	Prohibited	Available	Util%
CLB LUTs*	7866	0	0	1182240	0.67
LUT as Logic	7866	0	0	1182240	0.67
LUT as Memory	0	0	0	591840	0.00
CLB Registers	1521	0	0	2364480	0.06
Register as Flip Flop	1521	0	0	2364480	0.06
Register as Latch	0	0	0	2364480	0.00
CARRY8	1294	0	0	147780	0.88
F7 Muxes	1	0	0	591120	<0.01
F8 Muxes	0	0	0	295560	0.00
F9 Muxes	0	0	0	147780	0.00

Csynth

== Utilization Estimates

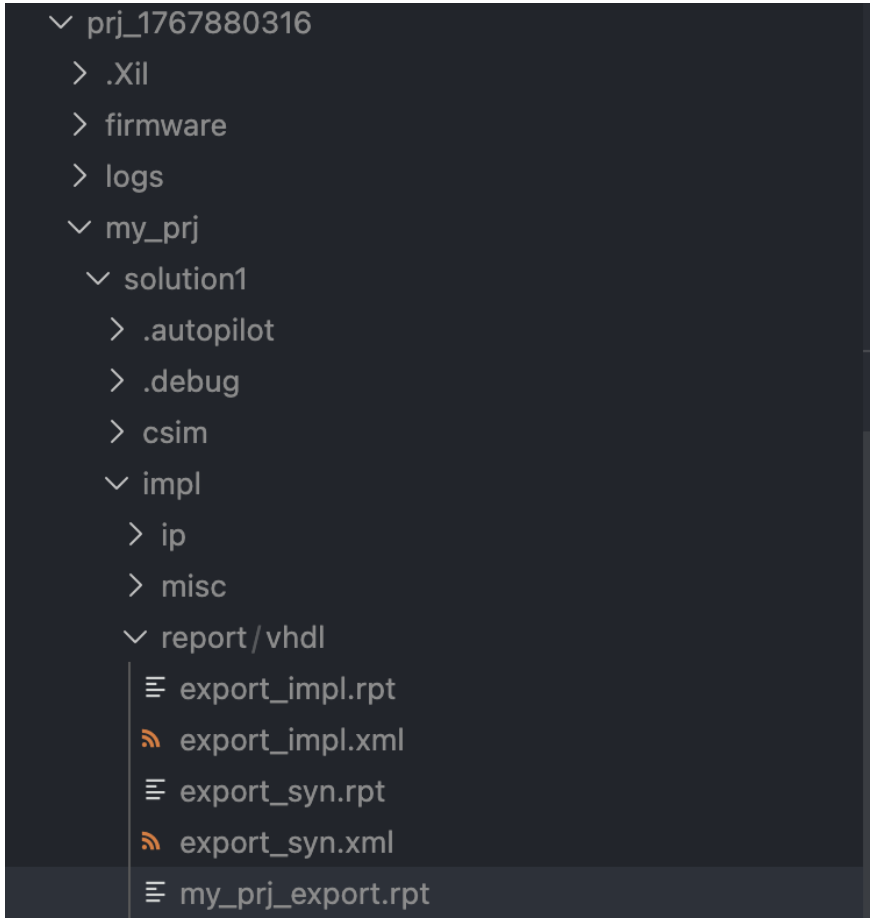
* Summary:

Name	BRAM_18K	DSP	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	2	-
FIFO	-	-	-	-	-
Instance	-	-	2181	64588	-
Memory	-	-	-	-	-
Multiplexer	-	-	-	-	-
Register	-	-	3	-	-
Total	0	0	2184	64590	0
Available SLR	1440	2280	788160	394080	320
Utilization SLR (%)	0	0	~0	16	0
Available	4320	6840	2364480	1182240	960
Utilization (%)	0	0	~0	5	0

FPGA design flow

- We can also do a standalone place and route where the netlist of the synthesis is mapped to a device

- Also “out-of-context”, no surrounding FW




```
#=== Post-Implementation Resource usage ===
SLICE:            0
LUT:           7708
FF:            1521
DSP:              0
BRAM:             0
URAM:             0
LATCH:            0
SRL:              0
CLB:           1509

#=== Final timing ===
CP required:                    5.000
CP achieved post-synthesis:     1.917
CP achieved post-implementation: 2.678
Timing met
```

FPGA design flow

- Now we have our HDL we can implement it for our chip
- Do a synthesis step, this synthesises the whole FW chain including any board infrastructure, I/O, clock management etc.
  - We can now do an RTL simulation
  - Simulates every signal a clock tick at a time
  - Needed for synchronisation debugging

FPGA design flow

- Now we can implement the design on the desired hardware
- This takes our netlist from the synthesis and places it on the hardware itself
- Part of this process is a “place and route”
- A process that takes ~ hours and has some level of randomness, one build one day might not be the same another day

FPGA design flow

- Timing closure is a large part of FPGA implementation
- Can my signal get from A to B in the time that I have set
- We talk about “slack” in timing
  - If my clock is 2.7 ns and my signal takes 1 ns to get from A->B I have 1.7ns of slack
  - If my signal takes 3 ns to get from A->B I have -0.3ns of slack
  - My “Worst Negative Slack” is the largest negative value of slack I have, the worst path that doesn’t meet timing
  - Even if your design had a csynth fmax of 500 MHz and the standalone place and route said timing was met, your final design may still have timing issues in that module

Solving timing issues

- Adding registers between algo blocks is a quick way of solving issues
- Placing FW in specific areas of the chip will help -> avoiding SLR crossings without first registering intermediate results
- You can ask the implementation tool to “try harder”

Thinking about designing FPGA FW

- You are not designing an algorithm, you are designing a processing system, think about how your data flows through the system
  - The fancy ML system that works in python may not work on an FPGA simply because all the data is not available at the same time
  - Understand the constraints of the system and the data flow in it, then start designing an algorithm

Thinking about designing FPGA FW

- Make algorithm blocks
  - Divide processing into smaller blocks
  - Allows swapping of algorithms without redesigning whole systems
  - Generic interfaces are your friend

```
-- ----------------------------------------------------------------------------
  LinksToTracksInstance : ENTITY work.TrackWordToVertexType
  PORT MAP(
    clk           => clk ,
    TrackWordsIn  => LinksIn ,
    VertexPipeOut => InputPipe
  );
-- ----------------------------------------------------------------------------
-- ----------------------------------------------------------------------------
  VertexDistributionInstance : ENTITY work.VertexDistribution
  PORT MAP(
    clk           => clk ,
    VertexPipeIn  => InputPipe ,
    VertexPipeOut => DistributedPipe
  );
-- ----------------------------------------------------------------------------
-- ----------------------------------------------------------------------------
  VertexFanoutInstance : ENTITY work.Fanout64_256
  PORT MAP(
    clk           => clk ,
    VertexPipeIn  => DistributedPipe ,
    VertexPipeOut => DistributedFannedPipe
  );
-- ----------------------------------------------------------------------------
-- ----------------------------------------------------------------------------
  HistogramInstance : ENTITY work.Histogram
  PORT MAP(
    clk           => clk ,
    VertexPipeIn  => DistributedFannedPipe ,
    VertexPipeOut => HistogramPipe
  );
-- ----------------------------------------------------------------------------
```

Thinking about designing FPGA FW

- Don’t be fooled by reports
  - The csynth report is a guideline, don’t rely on it, and don’t publish it
  - Just because the csynth told you you have an Fmax of 500 MHz does not mean your FPGA FW will meet timing at 360 MHz

Introduction to hls4ml

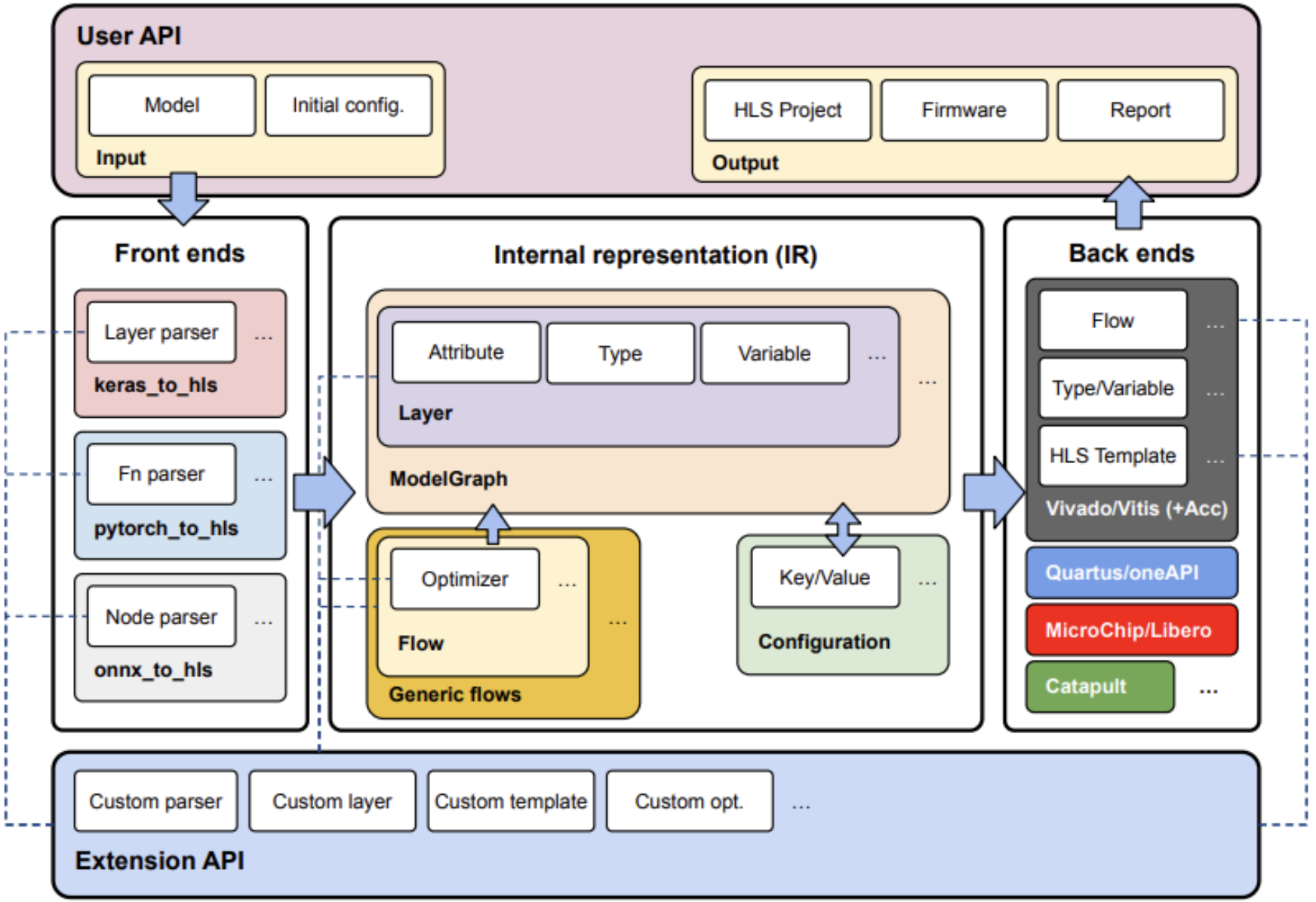
Christopher Brown
Epigraphy Kickoff meeting
12/01/25

Machine learning

- ML models are like onions, they have layers
  - We can peel them apart and each layer from an operation point of view looks pretty similar, I multiply my inputs together and push them through a function to give a result
  - We can make generic code for each of these layers in python
  - Let's make generic code for each of these layers in HLS

hls4ml

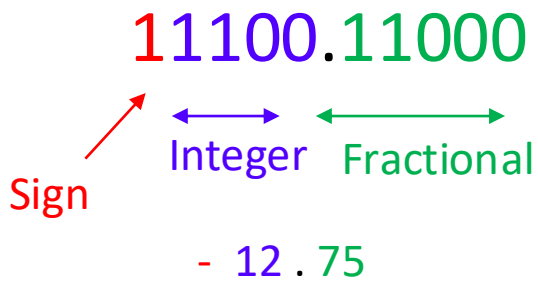
- Hls4ml is a generic framework for taking neural networks and making the hls for them
- Front end agnostic, can take Keras, pytorch QONNX
- Back end agnostic, vivado HLS, vitis HLS, Intel HLS, oneAPI, even ASIC
- Wide ranging support: Dense, Conv, Recurrent, some Graph
- Configurable: Can control the hls at a low level e.g. precision or parallelism


User API
Model
Initial config.
Input
HLS Project
Firmware
Report
Output
Front ends
Layer parser
keras_to_hls
Fn parser
pytorch_to_hls
Node parser
onnx_to_hls
Internal representation (IR)
Attribute
Type
Variable
Layer
ModelGraph
Optimizer
Flow
Generic flows
Key/Value
Configuration
Back ends
Flow
Type/Variable
HLS Template
Vivado/Vitis (+Acc)
Quartus/oneAPI
MicroChip/Libero
Catapult
Custom parser
Custom layer
Custom template
Custom opt.
Extension API

An aside on datatypes

- In FPGAs we don't work in floating point mathematics, we have to set where our integer and fractional bits are, so called "fixed point numbers"
- Every number has a integer set of bits i, a fractional set of bits f, and a sign bit s, total number of bits = 1 + i + f
- Integer: convert binary to integer direct
- Fractional: converted fractional bits * 2^{-f}

Link to vitis hls docs



Minimum possible value: $-s*2^{(i-1)}$
Maximum possible value: $2^{i} - 2^{-f}$
In steps of 2^{-f}

In hls we have the ap_fixed datatype

Where ap_fixed<10,5> represents a fixed point number with 10 total bits, 5 integer bits so 4 fraction bits where the min value is -16, max 31.8275 is and steps of 0.0625

hls4ml key parameters

- Clock period: in ns what is you clock period, e.g. 2.7 for 360 MHz clock

IO Type

- io_parallel: wires output of one layer into the input of the next, maximum throughput, minimum latency suitable for small models

- io_stream: Adds a FIFO buffer between each layer, single layers may have II > 1 so need buffering before. An additional optimizer can be used to optimise the depth of these buffers

Precision

- Can set hls4ml to be model or layer wise
- Can set an overall model precision
- Or single layer precision
- Typically use the automatic precision decided by hls4ml
- Can manually set the input and output quantisation, especially useful if plugging the model into a generic block with set input and output

CMVM

- Most operations in an ML layer are a constant matrix vector multiplication CMVM

- We have different ways of controlling this operation

Strategy

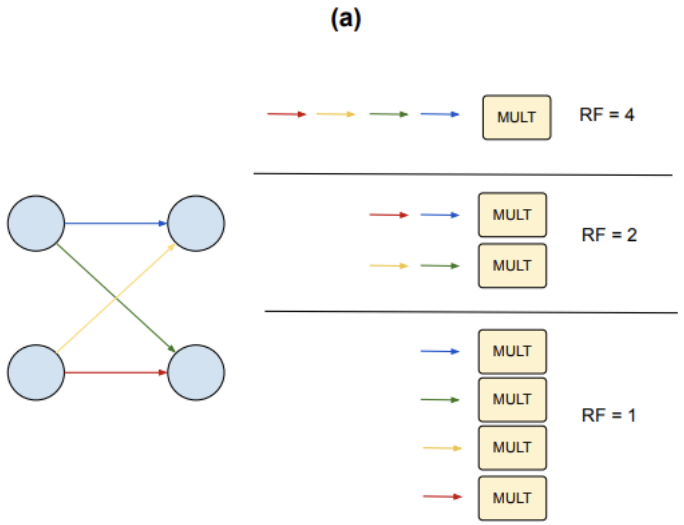
- Resource: Stores model weights in BRAM portioned into RF blocks so II=RF. Increasing RF decreased resources. Good for big models. (There is also a resource_unrolled for vitis only better for structured pruning models)

- Latency: suitable for small models with low inference latencies. Unrolls CMVM, RF won't necessarily change resources but dictates the II

- distributed_arithmetic: decomposes matrix vector multiplications into an adder with shift and add operations. Generally lower resources and latency but does not support RF > 1

Reuse factor

- Controls the parallelisation of the model
- Weight matrix is M x N, number of multiplications = M x N / RF



Parallelization factor

- Not the same as reuse factor
- Used in the case of identical matric vector multiplications on different inputs e.g. in a convolutional neural network
- Parallelises the total number of CMVM operations whereas RF parallelises the CMVM operation itself
- Higher PF II decreases, resources goes up

- We will see the impact of all these factors in the tutorial

Backup

FPGAs vs CPUs vs GPUs

	FPGA	**CPU**	**GPU**
Latency	Fixed	Variable	Variable
Clock Frequency	100s MHz	1s GHz	1s GHz
Processing time	10s ns	1s microseconds	1s microseconds
Parallelism	As much as you like	10s cores	100s cores
Specialism	Throughput	Bit of everything	Vectors
Programming difficulty	Hard	Easy	Easy (ish)
Power usage	10s W	100s W	100s W
Throughput	1s Tb/s	10 Gb/s	100s Gb/s