Introduction and Lab1

Aaron Zhao, Imperial College London

General Introduction

Introduction - Myself

My name is Aaron, and

- My research at college looks at Deep Learning (DL) Systems, generally focusing on efficient AI - how to make AI models more efficient in terms of computation, energy, and cost, either by improving the algorithms, or by co-designing the hardware and software together.
- I co-founded sequrity.ai, a startup that focuses on AI security and bringing security by design to AI agents.

My email is a.zhao@imperial.ac.uk, and my office is 903.

Introduction - Myself

- **Automatic Co-designing AI Systems with MASE**

  MASE aims to provide a unified representation for software-defined ML heterogeneous system exploration.

- **Beyond Structure Data**

  Investigate on unstructured and multimodal data, such as graphs, hypergraphs and combinatorial complex.

- **Efficient AI**

  Different algorithmic aspects of efficient AI, including efficient training, efficient inference, efficient model search and efficient model deployment with state-of-the-art GenAI models (eg. language and diffusion models).

Introduction - Teaching and RAs

The course is taught by me and Sarim Baig:

- I cover the FPGA parts
- Sarim covers the software parts

This course is also supported by the following GTAs and UTAs:

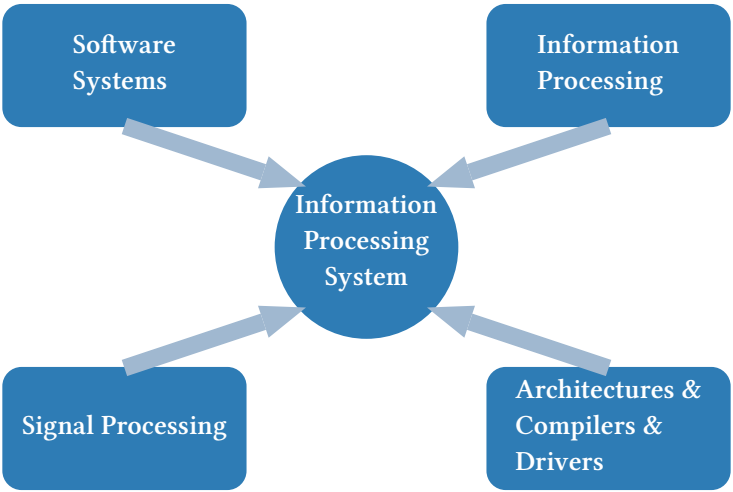
- Cheng Zhang
- Kevin Lau
- Eleanor Clifford

Introduction - Where to find stuff?

Everything is online:

- https://github.com/DeepWok/pynqchat/tree/main
  - Do not push to this repository
  - You can fork it and push to your own repository
- If you are new to git and github and do not understand what are these jargons – push and fork
  - check this link: https://www.youtube.com/watch?v=nT8KGYVurIU&ab_channel=TheCodex

Objectives and delivery



Objectives and delivery (ii)

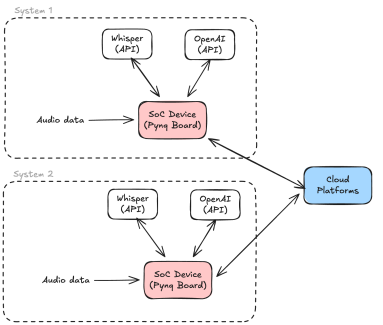
- Bring together theory and application from other modules
- Create an information processing system
- Project-based learning and integration of knowledge

Intended learning outcomes

- Design an information processing system that
  - Captures
  - Analyses
  - Manages
  - And outputs signals
- Implement an information processing system using a combination of software, hardware, networks, and databases
- Optimise a system to achieve given performance or quality targets
  - Latency
  - Throughput
  - Energy efficiency
  - Hardware resource usage

Let's be more specific: design an IoT system

- Nodes for local (signal) processing of information
  - audio data in this case
- Communication to a server
- Integration with a database
- Adapt processing in nodes



Let's be more specific: the development board

The PYNQ Z1 board

- FPGA (Xilinx/AMD Zynq-7000 SoC)
  - around 10K logic slices
  - around 600KB Block RAM
  - around 200 DSP blocks
- a 650MHz dual-core ARM Cortex-A9 processor
  - decent performance, supports full system Linux
- IO and connectivities
  - 3.5mm audio jack
  - Pmod connectors, arduino headers (good for later projects)
  - Ethernet port
  - Video capabilities (HDMI in/out)
  - User IOs like buttons, switches, LEDs

Let's be more specific: AWS DB

- Instantiate and use a database on the cloud
- Communicate from the host PC to the database through network

Structure and dates

- Phase 1 - Training (Week 2 - Week 5)
  - Lab based
  - with GTAs and UTAs for Q&A
- Mid-term Assessment (20%) (Week 6)
  - Lab orals
  - Show your engineering (a working system)
  - Show your improvements (up to 10 marks extra, on a qualitative basis)
    - what optional tasks have you done
    - optimization on end-to-end latency of the flow
  - Assessed as a group
- Phase 2 - Group project (Week 7 - Week 10)
  - Functional requirements
  - Non-functional requirements
- Final Assessment (80%) (Week 11)

Phase 1 - Training (the PYNQ Chatbot)

- Week 2 (Aaron)
  - Lab 1: Introduction to the PYNQ board. Install tools and learn how to program the device
  - Lab 2: Audio Processing: handling of the input audio and basic filtering
  - Understanding the overall system architecture, and basics in FPGA programming and designs
- Week 3 (Aaron)
  - Lab 3: Building the Chatbot: integrating Whisper and GPT models
  - Understanding how to integrate AI models into the system, basics on ASR, TTS and LLM-based inference systems.
- Week 4 (Sarim)
  - Lab 5: Create a remote server in AWS and run a custom service
- Week 5 (Sarim)
  - Lab 6: Create a remote database and perform queries

Phase 2 - Team project

General idea

- Local node needs to talk to a server (on the cloud).
- Server needs to talk back, the information needs to propagate back.

Elaboration

- Try to see how nodes can affect each other.
- Detect events, and change the processing in the nodes through a centralised server.

Requirements

- Log your events, or perform actions on the events.
- Detailed functional and non-functional requirements will be communicated

You do not have to build another chatbot system!

Logistics

- Lectures (Every Weeks 1-5, then in ad-hoc basis as needed)
- Groups and Communications
  - Groups of 5 or 6, have to all come from either group A or B.
  - If you cannot find a group, we will make one for you
  - You should start making a group now, deadline for this is 13th January
- Course Material
  - Teams
  - GitHub repository: https://github.com/DeepWok/pynqchat/tree/main

An introduction to FPGAs

Why an FPGA is an interesting device to consider

FPGA sells more than 40 million units per year, and is used in many applications, and we have the following vendors:

- AMD (Xilinx): around 50% of the market
- Intel (Altera): around 30% of the market
- Microsemi
- Lattice
- and more

Why an FPGA is an interesting device to consider

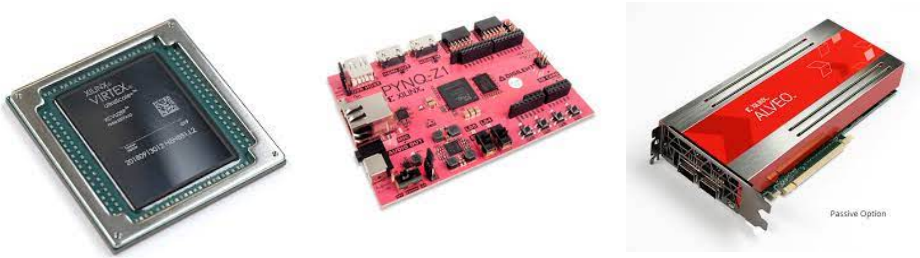


Figure 1: Xilinx Virtex chip (left), Xilinx Pynq Dev Board (middle), Xilinx Alveo accelerator card (right)

Similar to all other chips, FPGAs can be used in many applications and exist in many forms, such as on a development board (eg. Pynq), or as an accelerator card (eg. Alveo).

The growing need of energy efficient computing

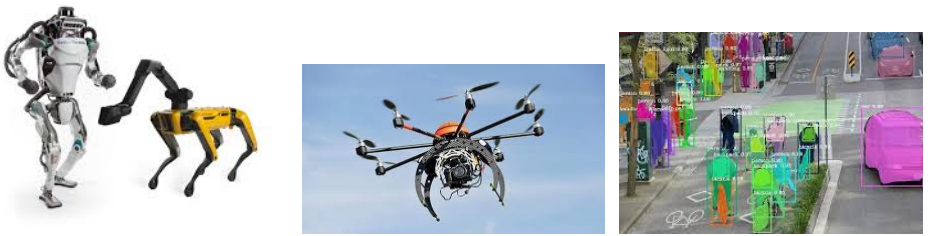


Figure 2: Robotics (left), UAVs (middle), Vision systems in cars (right)

There is now an increasing need for high-performance systems on low power systems (eg. robotics, UAVs, vision systems in cars).

The growing need of energy efficient computing



Figure 3: Bio-computing (left), large-scale simulation such as weather prediction (middle), AI computing (right)

There is now an increasing need for high-performance systems on large-scale workloads too. This is normally in large-scale data centres, where energy efficiency is a key concern.

The “ideal” computing device

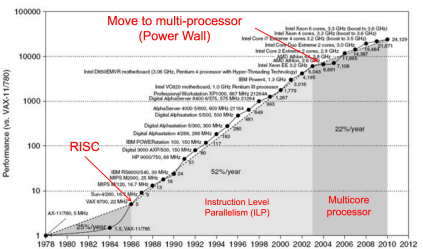
- Energy efficient
  - Betters Ops/Watt/Second/Area
- Cost efficient
  - Better running cost, somehow it is the same as energy efficiency
  - Better manufacturing cost
    - Die area
    - Yield
    - Time to market
    - nm node

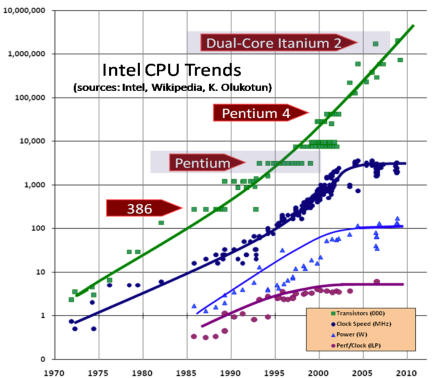
Approach 1

- Design or write more “efficient” algorithms
  - Better data structures
  - Better software engineering practices
  - Better compilers
  - Better hardware
- Use approximations
  - for instance, quantization or sub-sampling
- Consider the architecture of the system and optimize your program

Approach 2

Do nothing. Just wait for the next generation processor.





No more free lunch.

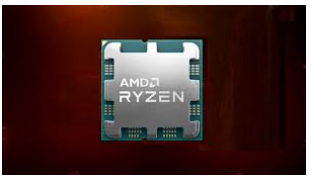
Approach 3

Use heterogeneous systems:

CPUs combined with other hardware (eg. parallel architectures, or ASICs) on the same SoC.

- Multi- or even Many-core CPUs (AMD Ryzen Threadripper PRO 7995WX, 96 Cores, 192 Threads 2.5GHz and max 5.1GHz)
- Graphic Processing Units (Nvidia H100, 640 Tensor Cores, 128 RT Cores, 80 Streaming Multiprocessors (SMs) and 18,432 CUDA cores)
- Field Programmable Gate Arrays (around 2M Logic Elements/LUTs, around 5K DSPs)

Approach 3 (ii)



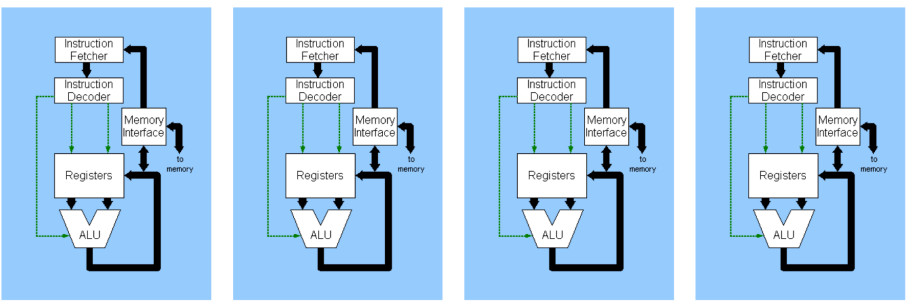
The flexibility and performance trade-off

- The left-hand side may have better power efficiency on the specific tasks.
- The right-hand side is more flexible and requires shorter dev cycles and efforts.

Benefits come from customising the hardware to the application, and also by tuning your application for the hardware, this is also known as **software-hardware co-optimization**.

Device comparison: multi-core CPUs

- Each core is fairly powerful and runs at a very high frequency.
- Complex memory hierarchy, eg. L1, L2, L3 caches.
- Up to linear speed up (extremely optimistic!).

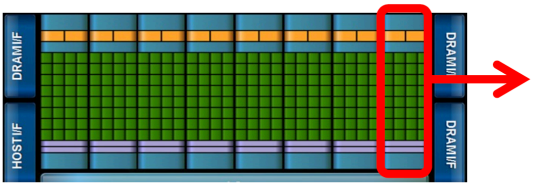


Device comparison: GPUs

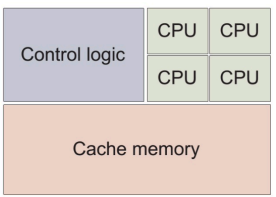
- Many light-weight thread processors (SMs, in Nvidia world) => Hide memory latency.
- All thread processors execute the same sequential code.
- SIMD architecture
  - massive data parallelism.
  - optimized memory access

Existing GPUs take advantage of the SIMD architecture, and are optimized for massive data parallelism. They also take over the top manufacturing process (eg. 3nm at TSMC). They are currently dominating the AI computing market.

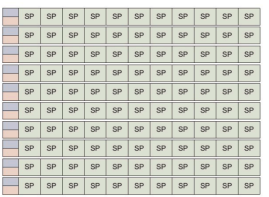
Device comparison: GPUs (ii)



1 multiprocessor = 32 thread processors



Multi-core CPU

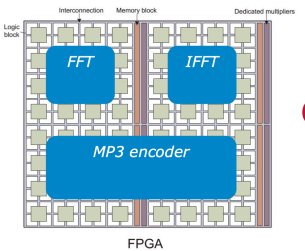


GPU

,

Device comparison: FPGAs

- (Re-)programmable digital hardware – can implement any digital circuit.
- Can exploit low-level pipeline parallelism Logic blocks evaluate simple Boolean functions.
- Interconnection resources connect blocks to implement complex systems.
- One way to understand is this is comoputign in “space” rather than “time”.



Computation in space

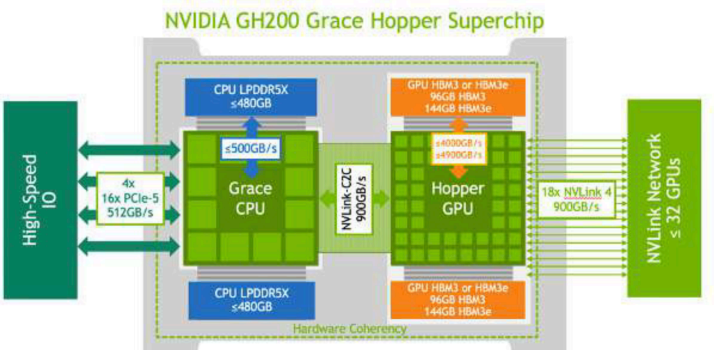
,

What price do you think we are paying for this reconfigurability?

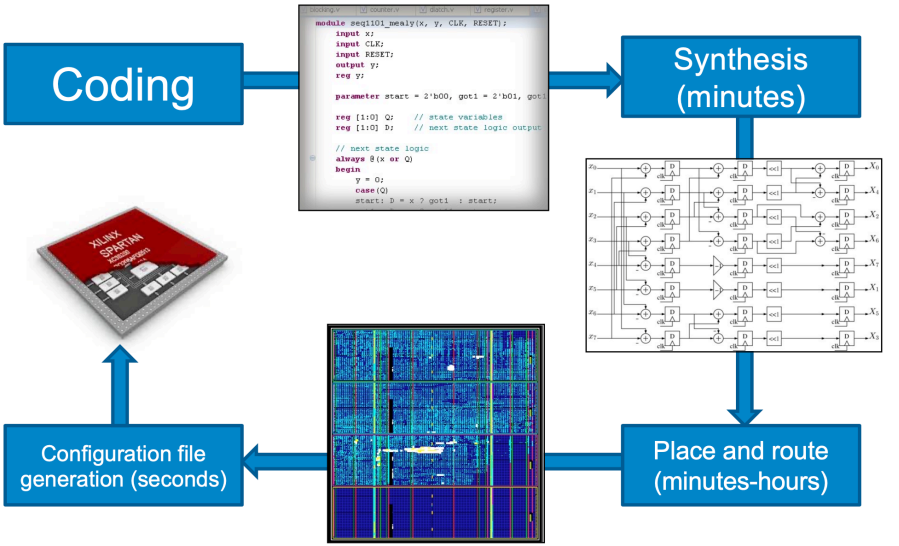
Heterogeneous computing

Normally in today's market, we see a combination of CPUs, GPUs, and FPGAs in the same system. This is known as a heterogeneous system. And this integration can actually happen at various granularities:

- Intel Xeon Gold 6138P with Arria 10 FPGA: server-scale CPU with FPGA
- NVIDIA GB200 architecture: Grace CPU with Hopper GPU



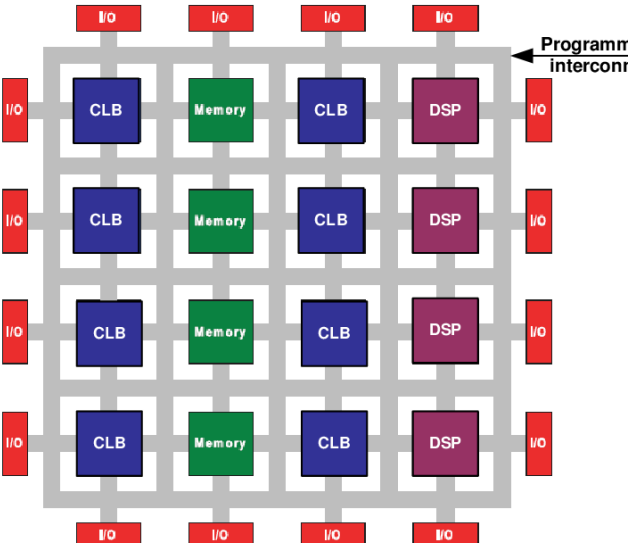
FPGA design flow



Field Programmable Gate Arrays (FPGAs)

- Xilinx (now part of AMD) is the first to introduce SRAM based FPGA using Lookup Tables (LUTs)
- Components
  - Configurable Logic Block (CLB)
  - Input/Output Blocks (IOBs)
  - Programmable Interconnects
  - DSPs
  - Block RAMs

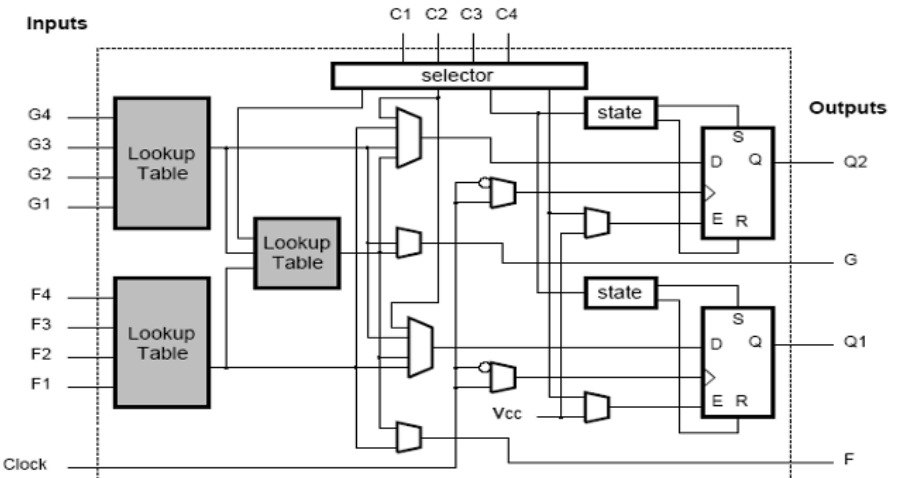
Field Programmable Gate Arrays (FPGAs) (ii)



CLBs

- Each Configurable Logic Block (CLB) has 2 main Look-up Tables (LUTs) and 2 registers.
- The two LUTs implement two independent logic functions F and G.
- Shown here is the CLB for Xilinx XC4000 devices.

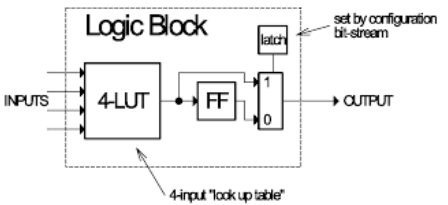
CLBs (ii)

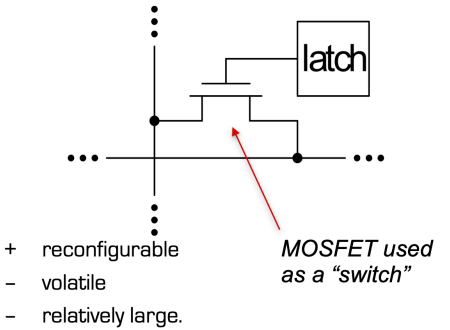

Inputs
C1 C2 C3 C4
selector
G4
G3
G2
G1
Lookup Table
Lookup Table
F4
F3
F2
F1
Lookup Table
Clock
state
S
D
Q
E
R
Q2
G
state
S
D
Q
E
R
Q1
Vcc
F
Outputs

LUTs

- LookUp Table (LUT) is implemented using latches
  - 4-LUT (i.e. 4-input LUT) implements any truth table with 4 inputs, this constructs combinatorial logic functions
  - Requires 24 storage elements, each implemented with a latch (similar to a flip-flop, but half the size roughly, 1-bit memory)
  - Multiplexer select one latch to output
  - “Configuration bit stream” is loaded under user control

LUTs (ii)

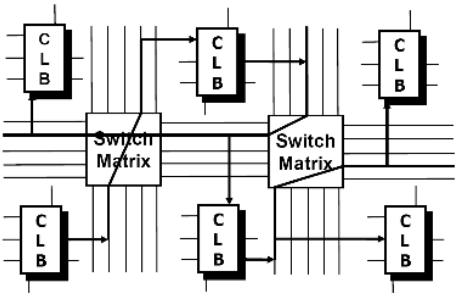




Typical interview questions: how many 4-LUTs are needed to implement an 8-bit adder?

Programmable interconnect

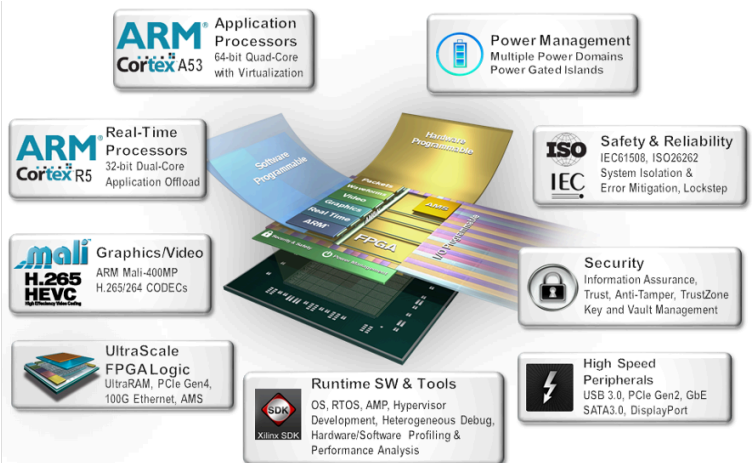
- Switch-box provides programmable interconnect
  - Local interconnects are fast and short
  - Horizontal and vertical interconnects are of various lengths



End of the Dinosaurs Age

Modern FPGA devices –- heterogeneous

Pre-built ASIC components are already integrated



Questions?

Any questions?

An introduction to Lab1

What is in this lab?

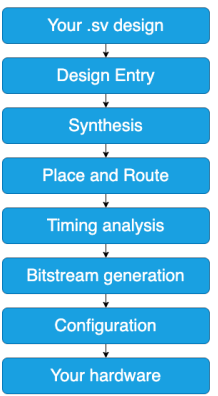
- Setting up the PYNQ board (the FPGA development board) and Vivado (the FPGA design software from Xilinx).
- Understand what is the system architecture:
  - the Processing System (PS)
  - the Programmable Logic (PL)
  - What is DMA?
  - What is AXI bus?
- Understand the software system:
  - How does software talk to hardware?
  - Low-level drivers (C++)
  - Mid-level wrapprers (Python)
- Toolflow to program the FPGA and run the software, understand the FPGA compilation process.
- Explore and test your design.

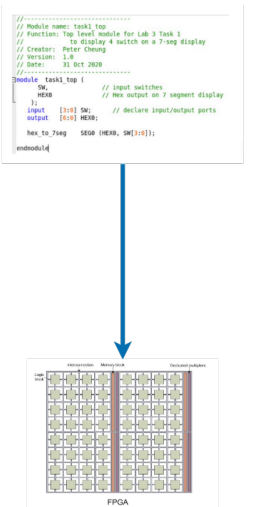
Setting up your environment

- Vivado (preferred setup is in the lab machine).
- I prefer you to do some parts of Lab1 ahead of time, because it involves downloading a lot of things: an OS image, or the Vivado software if it is on your own PC.
- Understand Jupyter Notebooks if you have not used them before.
- Be careful with your directory structure!

FPGA Compilation

You will have to go through a number of steps to map your design to the actual FPGA.





Lab 1 Overview

Part 1: Building an FIR filter using Vivado's IP creator

- Almost no hardware code writing
- Understand the FPGA toolflow
- Some Python coding
- Rely on Vivado's GUI and IP catalog
- Compare performance of hardware and software implementations
- Understand DMA transfers

Part 2: Create an `Merge` accelerator in hardware

- Some code writing in Verilog
- Some Python coding
- Building your own AXI4 peripheral through Vivado's IP creator
- Understand AXI4 interface

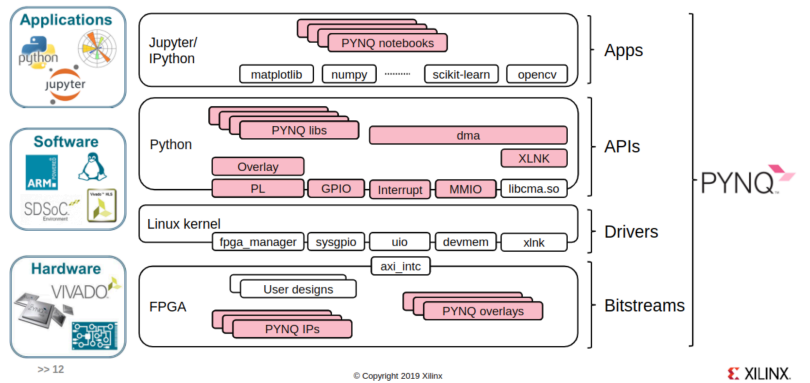
Lab 1: FIR Filter

Before we start, some jargons:

- PS: Processing System (the ARM cores)
- PL: Programmable Logic (the FPGA fabric)
- Zynq-7000: The SoC that contains both PS and PL, used on PYNQ Z1
- DMA: Direct Memory Access (data transfer engine between PS and PL)
- AXI: Advanced eXtensible Interface (the bus protocol used to connect PS and PL)
- DSP: Digital Signal Processing block (pre-built hardware block for multiply-accumulate operations)
- IPs: Intellectual Property cores (pre-built hardware blocks, eg. FFT, FIR filters, etc.)

Lab 1: FIR Filter: More on PYNQ

- PYNQ comes with full-system support - it runs Linux on the ARM cores
- PYNQ provides Linux drivers to talk to the FPGA fabric
- PYNQ provides an FPGA overlay to allow easy integration of custom hardware designs



Lab 1: FIR Filter: More fundamentals

What is a Hardware Driver?

- **A hardware driver** is a piece of software that allows the operating system and application software to communicate with hardware devices.
  - Think about what do you install when you buy a new printer or a new graphics card for your PC.
  - A driver is normally associated with a specific hardware device or a family of devices.
  - We use drivers in this lab to communicate with the custom hardware we build on the FPGA fabric from the ARM cores.
- **A Linux kernel** provides a standard interface for hardware drivers, which allows them to be easily integrated into the operating system.
  - Kernels provide APIs and mechanisms for drivers to register themselves, handle interrupts, and manage resources.
  - They are fundamentals for an OS

Lab 1: FIR Filter: Building an FIR filter

- One can build an FIR filter in software, but it is normally not very efficient.
- One can also build an FIR filter in hardware, which can be much more efficient.
- In the lab, we configure an FIR filter IP core using Vivado's IP catalog.
  - This means AMD (Xilinx) has already built a highly optimized FIR filter hardware block for us to use.
- **DMA** is a feature that allows hardware devices to transfer data directly to and from memory without involving the CPU.
  - This is important for high-performance applications, as it frees up the CPU to perform other tasks while data transfer is taking place.
  - We use DMA in this lab to transfer data between the PS and PL.

Lab 1: FIR Filter: Building an FIR filter

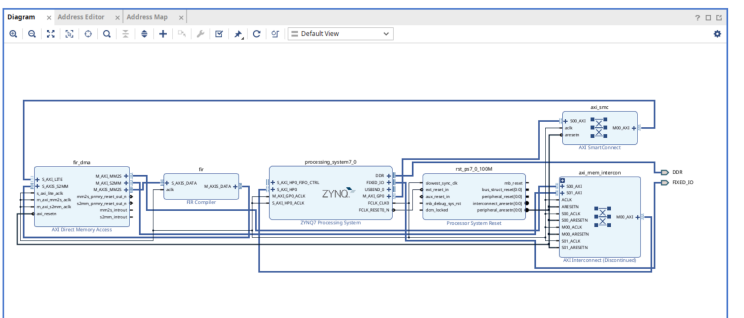
In simple words:

- Software layer: Python, through notebooks
- Middle layer 1: PYNQ libs in Python, mainly wrapped C++ drivers
- Middle layer 2: Linux kernel to manipulate drivers (C++)
- Middle layer 3: Your designed way to talk to hardware (eg. AXI4 peripheral in Verilog)
- Hardware layer: Your own FPGA design, built using Vivado

Lab 1: How to build an FIR filter in hardware?

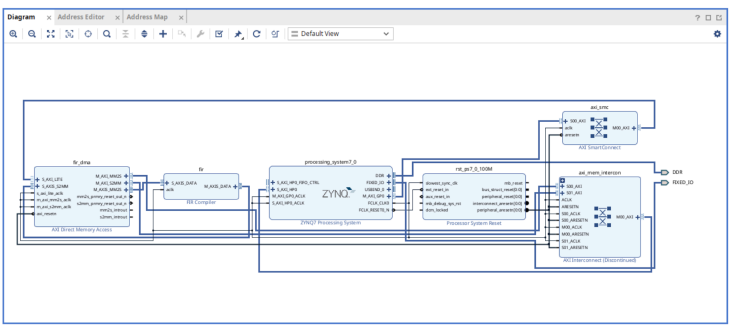
In a standard hardware practice, we write code in a Hardware Description Language (HDL), such as Verilog or VHDL.

In this first part of the lab, we will use Vivado's IP to compose a block design, this means you just click and drag to build your design, no HDL coding is needed (actually harder in my opinion!).



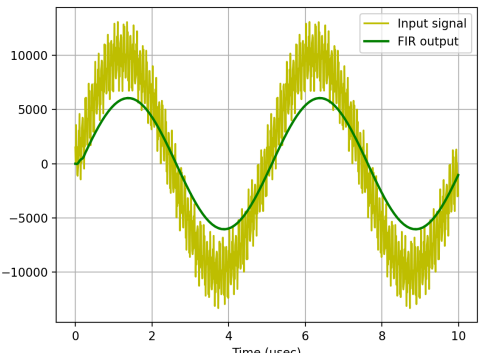
Lab 1: How to build an FIR filter in hardware?

- PS: Zynq-7000 Processing System (the ARM cores)
- PL: Programmable Logic (the FPGA fabric), used for the FIR
- The rest: DMA engines, AXI interconnects, etc. they connect the PS and PL together, manage data transfers, etc.



Lab 1: Direct speedup on FIR filtering

```
Hardware FIR execution time:  0.00954747200012207
Hardware acceleration factor:  9.249744037957297
```



Bonus question: do you think it would be faster than a pure C++ implementation on the ARM cores? Why?

Lab 1: Merge

In this lab, you will be asked to build a custom AXI4 peripheral that merges two input arrays into one output array.

```
input_1 = [1,3,5,7]
input_2 = [2,4,6,8]
output = [1,2,3,4,5,6,7,8]
```

Very easy task in software, but you will have to build it in hardware this time!

Lab 1: mergeCore.v

Basic Design

- 2 input FIFOs
- 1 output FIFO
- 1 Finite State Machine (FSM) to control the merging process
  - 5 states
  - IDLE, COMPARE, FLUSH_FIFO, WRITE, DONE

What defines a FIFO

- Full and empty signals
- Read and write pointers
- Data width and depth
- Synchronous or asynchronous

```
module fifo #(parameter DATA_WIDTH=32, parameter DEPTH=16) (
    input wire clk,
    input wire rst,
    input wire wr_en,
    input wire rd_en,
    input wire [DATA_WIDTH-1:0] din,
    output reg [DATA_WIDTH-1:0] dout,
    output reg full,
    output reg empty
);
    // FIFO implementation here
```

FSM

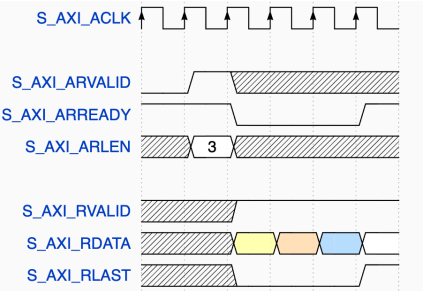
- A good habit is to define state using localparam
- Another good habit is to separate the state transition logic and state output logic, so that the code is more readable and maintainable.

```
// State encoding
localparam IDLE = 3'b000;
localparam COMPARE = 3'b001;
...
// State transition logic
always @(posedge clk) begin
    case (current_state)
        IDLE: begin
            if (start) current_state <= COMPARE;
        end
        ...
    endcase
end
// State output
```

FSM (ii)

```
always_comb begin
    case (current_state)
        IDLE: begin
          output <= 0;
        end
        COMPARE: begin
          output <= 1;
          ...
        end
        ...
    endcase
end
```

Lab 1: What is AXI?



This is the bus protocol that connects modules together.

- The first key requirement of any high performance AXI slave is that the ARREADY line must be high when the slave isn't busy.
- The transaction then starts with the request on the read address channel, as indicated by ARVALID. This will tell us the address we want to read, ARADDR, and the number of items to be read, ARLEN+1.

Lab 1: Merge: Summary

In this lab, you will pack your Verilog code into an AXI4 peripheral using Vivado's IP packager.

In that IP, you will also make use of Vivado's FIFO IP core for the two input FIFOs and one output FIFO.

Finally, you will integrate your IP into a block design, connect it to the ZYNQ PS, and generate the bitstream.