LELEC2531 Project

Context

This project aims to integrate an accelerator into the DE0 Nano and Raspberry Pi architecture, as illustrated in Figure 1.

The project is designed to give you practical insights about the entire course stack, with a focus on: **(1) pipelined processor** architecture, **(2)** the various **abstraction levels** covered in the course, **(3)** system-level **bus transfers**, and **(4) multi-level hardware** design.

Before beginning the project, it is essential to have a proper understanding of system-level data transfers, particularly memory access. We strongly recommend reviewing the material from **Labs 4 and 5**, as well as the architecture illustrated in Figure 1. To ensure you have a thorough grasp of Figure 1, you should be able to answer the following questions:

1. What is the purpose of each SystemVerilog .sv file?
2. How does the pipelined processor determine where to read or write data?
3. Where are the memory addresses defined?

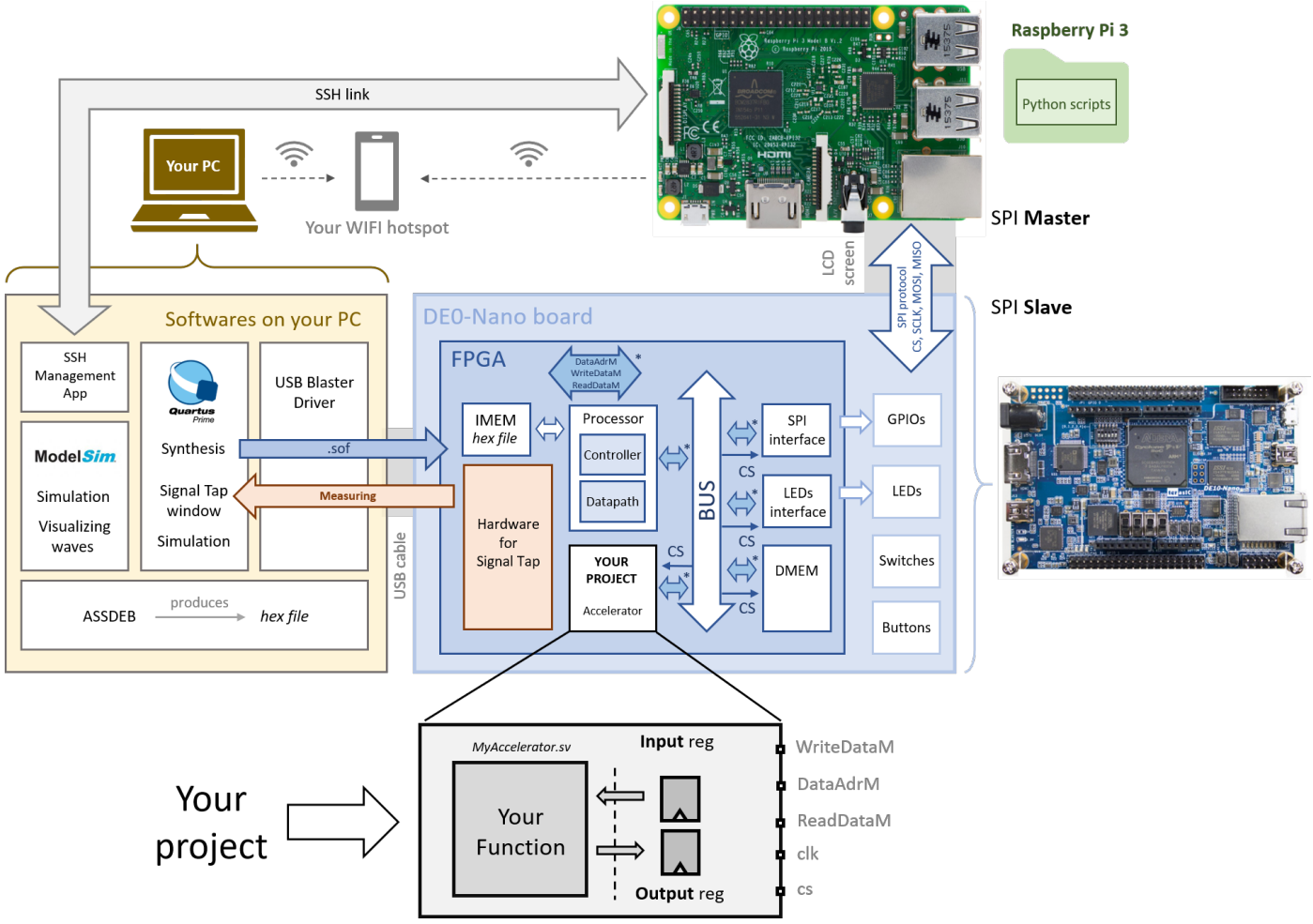

Figure 1: Full LELEC2531 setup including Raspberry Pi and DE0 nano boards, highlighting your project.

1 Adding a dedicated accelerator

Why do we need an accelerator ?

Let's consider a scenario where we want the pipelined processor to perform **computations that are not natively supported by its instruction set**. What options does the designer have? The most straightforward approach would be to write a **program in assembly language**. However, since the instruction set does not include a dedicated instruction to perform specialised tasks, you would need to break down complex mathematical operations into a series of basic instructions such as additions, subtractions, and multiplications. While this method will eventually produce the correct result, it requires writing a **complex assembly program** and consumes **a significant number of processor cycles** to complete.

Now imagine needing to make numerous computation-intensive tasks. The total number of processor cycles required would scale proportionally, leading to **excessive computing time and power consumption**. Moreover, while the processor is engaged in these operations, **it cannot perform other tasks**, leading to a **bottleneck**. Clearly, a more efficient solution is needed.

You might consider **changing the processor's architecture** to include a custom instruction. Although this approach is **theoretically possible**, it poses a **significant issue**. In real-world situations, processors are typically licensed as **commercial intellectual properties** (IPs) by companies such as Intel, AMD, or ARM. Thus, these designs are provided as closed systems, or "black boxes" that do not permit modifications to their internal architecture.

To cope with these issues, we can directly design a specialized hardware block to perform this operation. This hardware block is called "accelerator". Doing your own accelerator has the following advantages:

1. **Performance improvement:** It executes computations in **fewer cycles** than a general-purpose processor.
2. **Parallel operation:** The accelerator manages tasks while the **processor remains available** for other operations.
3. **No processor modification:** The processor's **architecture stays unchanged**, avoiding complexity and licensing issues.

Accelerators are commonly used to offload demanding tasks from the processor. Examples include image processing tasks (e.g., those in graphics cards), cryptographic operations, and the encoding and decoding of compressed files, among many other compute-intensive applications. This approach not only improves efficiency but also illustrates the versatility of hardware-software co-design in modern systems.

Specific to your project

Your **starting point** will be the folder containing the pipelined processor architecture introduced in **Lab 5**. You will **integrate the accelerator attributed to your group** into the **system bus** by assigning it a specific **memory-mapped address range**. Accessing the accelerator (whether to read, write, or trigger computations) will operate under the **same principles as peripheral access** (e.g., the SPI unit). During the MEM stage of memory instruction execution, if the target address falls within the accelerator's range, the pipelined processor will interact with its interface registers instead of the DMEM. This mechanism allows you to **send input data** to the accelerator, **retrieve its output**.

Minimum interface requirements for the accelerator module :

1. **Input Register:** One 32-bit register to hold the input data for computations.
2. **Output Register:** One 32-bit register to store and provide the computation results.
3. A **clock** signal.
4. A **chip** select signal.
5. A **reset** signal.

Any entries added to this module definition must be justified during the examination. The specific function you need to implement has been assigned to you on Moodle.

Once your hardware accelerator is functional, we ask you quantify its performance. The performance metrics that we ask you to use are: (1) FPGA hardware usage (number of logic, LUT, memory blocks used), (2) area occupancy and (3) its estimated power consumption.

2 The day of the examination

You are expected to:

1. Demonstrate the accelerator operation:
   - Send the **32-bit input data** to the accelerator **via SPI** by executing a **Python script** on your **Raspberry Pi**.
   - Retrieve the **output data** from the accelerator, then send it **to your Raspberry Pi** via **SPI**.
   - Confirm that the SPI transfer occurred by setting the appropriate **triggering condition in SignalTap**.
2. Assess your accelerator implementation:
   - **Review your code** with us during the presentation.
   - **Discuss the advantages** of integrating your accelerator into the overall system architecture. What is its performance in terms of the following KPI : (1) FPGA hardware usage (number of logic, LUT, memory blocks used), (2) area, and (3) the estimated power consumption. If specific performances are difficult to evaluate, please specify why.