H05D3A: Computer Architectures
Exercise sessions 1-2-3

Goal of these exercise sessions

These exercise sessions aim to help you build a good understanding of the basic RISC-V CPU architecture and the typical digital ASIC design flow. In the sessions, you will develop a complete processor in Verilog, going through simulation, synthesis, and place & route steps till eventually generating the GDSII file.

1) You start by completing the necessary components of a single-cycle processor and simulating it with a simple program. Then add the multiplication instruction support for.
2) Next, you will increase its performance by pipelining the processor, based on which you will learn how to run the full digital backend flow.
3) Afterwards, you will update your CPU architecture to handle different hazard situations.
4) In the end, you will use the knowledge you learned from the course to optimize the latency of running a matrix-matrix multiplication program.

Information about the tools

To complete the assignment, you will work with two kinds of tools:

One kind is for the functional correctness check, using **NC Verilog**.

Another kind is for the digital backend flow, for which we adopt the **OpenLane** flow, as shown below. The tools used in each separate step are highlighted with red boxes.
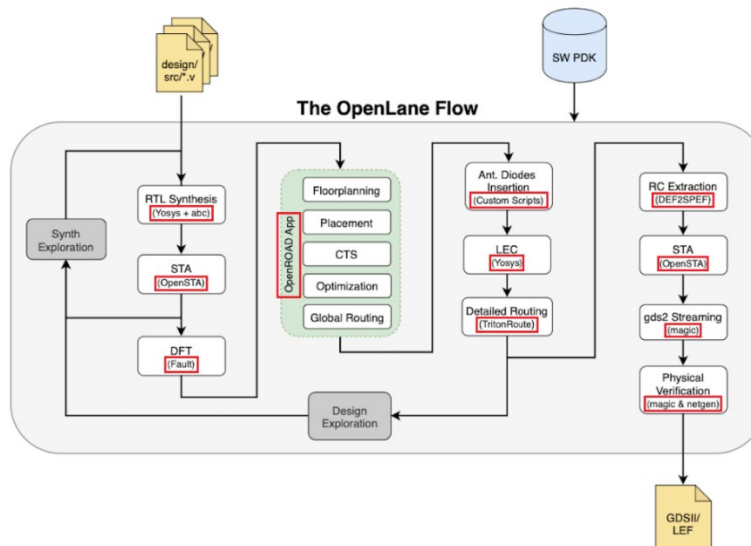
NC Verilog helps you to perform cycle-accurate simulation based on your testbench and HDL codes. OpenLane helps you to automated RTL to GDSII flow based on several components including OpenROAD, Yosys, Magic, Netgen, CVC, SPEF-Extractor, KLayout and several custom scripts for design exploration and optimization. You can find more information here: link. The 130nm SkyWater open-source PDK, SKY130, is used in the backend flow for this exercise.

These tools are set up for you.

To create your workspace, you have two options:

1. Download from the repository at https://github.com/KUL-ESAT-CA/CA_Exercise and use *git* as versioning tool of the development of the project (suggested).
2. Download from Toledo the file *CA_processor_design_2023.zip* Extract it to some location of your system.

The folders that compose our project workspace are:
- Verilog: All the Verilog files that define the logic of the processor.
- SIM: Files for running the simulation of the HDL code.
    - SIM/data: Subfolder that contains the assembly code to run.
- Backend: Files for running the backend flow for the processor.

For using the tools and completing the exercise sessions, it is recommended to have access to ESAT servers. You can either come to the ESAT computer room or connect remotely through SSH connection.

If you are not familiar with SSH connections and/or *git* versioning tool and want to use them, please check out the documents and guides provided for you in the course material.

RUN CYCLE-ACCURATE SIMULATION (FOR FUNCTIONAL CORRECTNESS CHECK):

To run the simulation of the processor:

- Go to the folder SIM. Run the command `source incisiv_15.20.058.rc`

- Still in folder SIM, run the command `make nc_gui` through a terminal (Optionally run the command `make nc` to use the tool without a graphical interface which checks the correctness of the program). This command will call all the HDL files declared in SIM/files_verilog.f and will execute the testbench RTL/cpu_tb.v. This testbench

    1) declares an instance of the cpu

    2) loads the instruction memory from the file SIM/data/imem_content.txt

    3) loads the data memory from the file SIM/data/dmem_content.txt

    4) starts the execution of the instructions.

- After running the command, the NC Verilog graphical interface will open. In the left part of the window the whole hierarchy of the design can be found.

- With the objective of displaying the internal signals for debugging, go to the target submodule and select it. Afterwards do right-click over the signals to display and select "Send to Waveform Window". Finally run the simulation with the button:

- To change the current executed program (to change instruction and data memory content) modify the hexadecimal files imem_content.txt and dmem_content.txt (contained in the folder SIM/data) with the instruction memory and data memory respectively. Several example programs can be found in the folder SIM/data/testcode. The files with the format *NAME_OF_PROGRAM_imem_content.txt* and *NAME_OF_PROGRAM_dmem_content.txt*, which contain the data for the instruction memory as well as the data for the data memory respectively, must be copied into the files SIM/data/*imem_content.txt* and SIM/data/*dmem_content.txt.* These last-mentioned files are the files effectively loaded by the simulation.
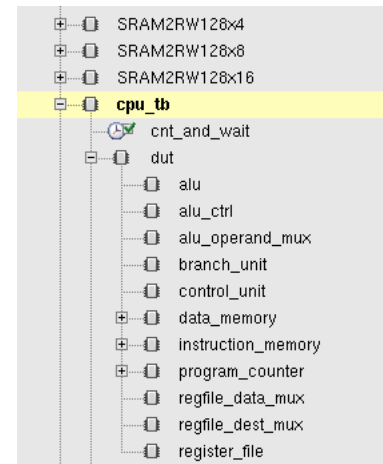
You can prepare the backend environment using the following steps:

- Go to the "Backend" folder, run the command: source setup.sh. If it is the first time you source it, it will take 5 mins to install the backend tools and jupyter by micromamba.
- In the terminal, type the command jupyter-lab. This should lead you to the website of Jupyter notebook.
- The Jupyter notebook is divided into 3 parts. The first part is the prerequisite of the scripts, like environment variables' settings, function definition etc. The second part is about the synthesis. You need to run synthesis with every implementation. The last part first leads you to the floorplanning step with visualization, and then goes through the whole backend flow. You will be able to see the layout of the chip.

Tasks to be done:

The datapath as well as its submodules have already been implemented for you in Verilog. However, the functionality of the processor must be completed by carrying out the tasks described underneath (1,2,3,4,5).

1. Single-Cycle processor

Complete the internals of the control unit to handle the following instructions: (BEQ, JUMP, LW, SW, ADDI as well as R-type ALU instructions called ' ALU_R'). Set the outputs of the control unit correctly depending on the fetched instruction, ensure the correct datapath behavior.

==> For debugging purposes, you can use the test code *simple_program* in the folder SIM/data. Next, the code *"MULT1"* executes the multiplication of 5 integers and sums all the results. Test this code to ensure that your design is correctly implemented. Then, fill the cycle counts into the report table.

==> At this step, you are also required to run the synthesis. You need to set up the backend tools and open the Jupyter Lab, then follow the instruction in synthesis stage. Fill in the timing and area info after synthesis into the report table.

2. Single Cycle processor with multiplier

The processor completed in the last exercise can execute multiplications through sums and shifting since there is no hardware support to carry out the multiplication in one cycle. With the purpose of boosting performance, your task will be to add hardware support for multiplication and evaluating its impact on performance (number of cycles to finish the run). The processor must be able to process the multiplication instruction MULT. The format of the instruction is the following:

0000001	XXXXX	XXXXX	000	XXXXX	0110011
[31:25] funct7	[24:20] rs2	[19:15] rs1	[14:12] funct3	[11:7] rd	[6:0] opcode

Three files might need to be modified for this purpose: *control.unit.v*, *alu_control.v* and *alu.v.* To test the performance and correctness of your design, load the assembly test code *"MULT2"* which now uses the implemented MULT instruction to carry out the same 5 multiplications and sums of the testcode *"MULT1"*.   After updating your design, run the simulation and synthesis.

- *(For you to think) Is it necessary to modify the control unit for the insertion of the MULT instruction?*

3. PIPELINED processor

In this part of the session, we will modify our processor to convert it to a pipelined implementation. For now, assume data and control hazards are solved by inserting NOP instructions.

For a processor with 5 pipelined stages (Instruction Fetch (IF), Instruction Decode (ID), Execute (EXE), Memory (MEM) and Write Back (WB)), identify which hardware resources belong to each of the mentioned stages, and which signals should go from stage to stage. Using Figure 4.31 of the book, try to match the HDL code with the Architecture and complete the tables underneath.

	IF	ID	EXE	MEM	WB
Datapath Resources					

	IF > ID	ID > EXE	EXE > MEM	MEM > WB
Signals				

Insert the pipelined registers where necessary, using the module *reg_arstn_en*. This module implements a register with variable width, which is set through the parameter DATA_W. See the inset on the right for an example for such a pipelined register between the Fetching (signals *signal_IF*) and Decode stage (*signal_ID*). Connect the enable signal of the pipelined registers to the global enable signal called *enable*. After updating your design, run the test code MULT*2* again and ensure the correct functionality.

```
reg_arstn_en#(
    .DATA_W(16)
) signal_pipe_ID_EX(
    .clk   (clk      ),
    .arst_n(arst_n   ),
    .din   (signal_IF ),
    .en    (enable    ),
    .dout  (signal_ID )
);
```

Afterwards, open the Jupyter Notebook to run through the complete backend flow. Try to understand the general idea of the flow. Note down the timing and area info after Logic Synthesis step. Then track the change of timing and area in the reports of subsequent steps, and reason for this change. Also, in the notebook, we provide you with the script to visualize the chip formation step-by-step.
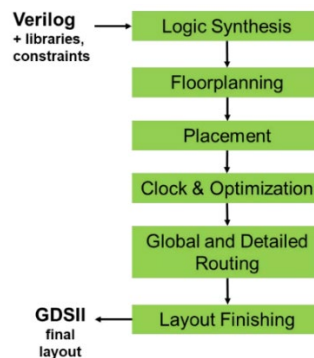
The general digital backend flow steps

4. PIPELINED PROCESSOR WITH DATA-HAZARD RESOLUTION

Since the pipelined processor implemented does not have support for handling data hazards, the execution of more complex code containing branches or data dependencies would not work properly.

Add hardware support for forwarding and stalling to achieve this functionality. After updating your design, run the test code MULT*3* to ensure the correct functionality.

Finally, open the Jupyter Notebook to run through synthesis steps and compare the results with the previous implementations. If you are interested, you can also run through the entire backend flow for better comparison (not required).

5. ADVANCED ACCELERATION

This last exercise aims at demonstrating how an architecture can leverage the structure of a program to speed up its execution. To do so, your will accelerate the execution of the following matrix-matrix multiplication:

$$\begin{pmatrix} 20 & 19 & 18 & 17 & 16 \\ 15 & 14 & 13 & 12 & 11 \\ 10 & 9 & 8 & 7 & 6 \\ 5 & 4 & 3 & 2 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \\ 10 & 11 & 12 \\ 13 & 14 & 15 \end{pmatrix} = \begin{pmatrix} 600 & 690 & 780 \\ 425 & 490 & 555 \\ 250 & 290 & 330 \\ 75 & 90 & 105 \end{pmatrix}$$

For this exercise, a baseline program (MULT4) is provided to execute this matrix multiplication. This program operates on two matrices I and W stored in row-major order and column-major order, respectively, and appends the result O in row-major order after the operand matrices in the data memory space.

Using your pipelined implementation as a basis, elaborate a solution to accelerate the executions of matrix multiplications (hints: Patterson-Hennessy Section 4.8 to 4.10, RISC-V specification chapter 17). Simulate your design to ensure correctness and check the latency (number of cycles to complete MULT4). Then, run the backend flow (synthesis required, floorplan & signoff optional) to get the timing and area information. In the end, you can visualize the generated GDSII file.

- The assignments must be completed in groups of **2 people**.
- After completing each one of the exercises the results must be shown to the TAs for its correspondent evaluation.
- You will hand in a small report which includes the performance of your processor after each step and some questions to answer.
- This project counts for **4 points** in the final H05d3A grade. The **report** and your **final processor design** need to be handed in through Toledo, following the below file structure:

```
./Group_X/
├── README.txt
├── Report.pdf
├── SIM_files
│   ├── files_verilog.f
│   └── mult4_imem_content.txt
└── Verilog_files
    ├── cpu_tb.v
    ├── RTL_SOLUTION1_simple_program_and_MULT1
    ├── RTL_SOLUTION2_multiplication_support_MULT2
    ├── RTL_SOLUTION3_pipeline_basic_MULT2
    ├── RTL_SOLUTION4_pipeline_hazard_MULT3
    ├── RTL_SOLUTION5_pipeline_creative_hazard_MULT4
    └── sky130_sram_2rw.v

7 directories, 6 files
```

  - Name your project as Group_X (X is your group number);
  - Put all the students' name and students' R-number of the group in README.txt;
  - You can also put anything that you think we need to know before running your project in the README.txt.

- The deadline of the project is **May. 19th, 2023**.