Project: GeMM Accelerator Design

CPAEP

Academic year 2025–2026

Submission deadline: **December 7th 2025, 19:00**

Instructors: Marian Verhelst, Arne Symons, Ryan Antonio, Yunhao Deng
Contact: `firstname.lastname@kuleuven.be`

This document describes the CPAEP project assignment. Section 1 defines the project goals, design task, and report questions. Appendix A helps you get the repository running. Appendix B provides a suggested step-by-step design workflow.

1 Project details

1.1 Objective and learning goals

This project is by pair (groups of 2), and the goal of this project is to design, implement, and analyze a parametrizable GeMM (general matrix multiplication) accelerator in SystemVerilog.

You will begin from a provided 1-MAC accelerator baseline and its supporting infrastructure. Your task is to architect a 64-MAC datapath capable of efficiently accelerating several matrix sizes, to design the corresponding address-generation and control logic, and to validate and analyze your design using simulations, waveforms, and basic coverage.

After completing the project, you should be able to:

- Translate GeMM requirements into a spatial accelerator architecture.
- Reason about latency, utilization, arithmetic intensity, and bandwidth.
- Design address generators and memory layouts that match a chosen dataflow.
- Discuss critical-path and pipelining trade-offs at a qualitative level.
- Use tests and simple coverage metrics to argue about verification quality.

1.2 Design task and constraints

- You are given a fixed hardware budget of **64 MAC units**.
- You are free to choose the array shape and how these MACs are organized (e.g., a 1D, 2D or 3D array, a hierarchical structure, or a more adaptive/versatile core).

- You may choose any data orchestration strategy for mapping operands into the MAC array.
- Different array organizations will favor different workloads and introduce varying implementation complexity.
- Your accelerator must support the following three GeMM workloads for operands $A \times B = C$:

Case	Matrix A size	Matrix B size	Matrix C size
1	4×64	64×16	4×16
2	16×64	64×4	16×4
3	32×32	32×32	32×32

- All input data (matrices A and B) must be stored in `int8`, and all outputs (matrix C) must use `int32`.
- Memory modules will be provided (see Appendix A.4) and include a parametrizable number of data width and memory depth.
- **The primary performance objective is to minimize the average normalized latency across the three cases.**
- **Normalize the latency of each case by the total number of MAC operations of the case before averaging**
- Achieving this objective requires high arithmetic intensity and strong spatial utilization across all workloads.

1.3 Provided code base

You will use an existing repository containing the following (more details are in Appendix A.1):

- A simplified yet realistic repository structure, organized in a style commonly used by digital designers.
- A set of structured test benches, including a main test bench and several unit-level test benches.
- A baseline single-MAC GeMM design with a reference datapath is offered.
- An accelerator-side FSM controller with counters and address-generation logic.
- Memory modules that emulate SRAM behavior, along with a baseline matrix data layout.
- A configurable MAC block that you may customize to meet your design requirements.
- Inline comments throughout the code base that provide guidance. You are expected to study the baseline implementation and adapt it to your chosen architecture.
- **This baseline serves as an illustration of how our testbench is constructed.**

- **Going through this design before starting yours to ensure your design is compatible with our I/O definition.**
- **You are free to design your accelerator as you see fit: You can choose to design a completely new accelerator or extend the design based on the baseline.**
- **We encourage you to choose the first option, as our baseline may not be enough for the fancy architecture you want to add.**

1.4 Required work and deliverables

You are expected to:

- **Datapath and architecture:** Design a 64-MAC GeMM array topology and interconnect. You can design your own MAC unit if you use another stationary strategy, but at most one multiplication can be done in each cycle.
- **Control and address generation:** Adapt or redesign the main FSM controller, counters, and address-generation units so they implement your chosen mapping and loop structure for all three workloads.
- **Memory and data layout:** Define how matrices A, B, and C are stored in memory, how the accelerator accesses this data (e.g., parallel and high-bandwidth patterns), and revise the layout to stream data efficiently into and out of the array. You can only use the SRAM macro we provided as the final data access point. **You are only allowed to change data width (bit-width) and the depth of the memory.**
- **Verification:** Extend and run unit tests, use the provided testbench to validate all three workloads in one simulation launch without resetting the dut, and use coverage reports or similar evidence to demonstrate that your design is reasonably well tested.

You will submit:

- **Design files:** Your complete repository containing all Verilog/SystemVerilog source files, along with any additional testbenches or scripts required to run your experiments.
- **Written report:** Using the **CPAEP project report template**, answer the design questions in Section 2 and summarize your design decisions, results, and reflections. More details in Section 2. You are encouraged to create your project report in the Overleaf hosted by ESAT.
- **Submission method:**
  - Submit your entire working directory as a `.zip` file in Toledo, named `cpaep_2526_lastname1_lastname2.zip`.
  - Include in the `README.md` clear instructions on how to run your code.
  - Include your final report PDF at the root of your repository, named `cpaep_2526_report_lastname1_lastname2.pdf`.
  - Also include the disclosure for use of GenAI form. Name it as `cpaep_2526_genai_form_lastname1_lastname2.pdf`

1.5 Conslutation time

During the course of the project, you are free to work on the project anywhere you want. But you can always consult the TAs at the times and place listed below:

- Official schedule:
  `https://onderwijsaanbod.kuleuven.be/2025/syllabi/e/H05H2AE.htm?ids=50448192`
- Monday (1/12): 13:30–15:55 (01.62)
- Wednesday (3/12): 8:00–10:30 and 16:00–18:25 (02.54)
- Thursday (4/12): 16:00–18:25 (01.54)
- Off-hour consultation is possible by appointment. Please email the TAs.

2 Expected reporting on your design

At the end of the project, you will submit a report together with your code. In this report, we expect you to include the items described underneath. Before starting your report, make sure you have:

- Followed the setup instructions in Appendix A to run the baseline design and tests.
- Worked through the design workflow in Appendix B to arrive at a stable, tested design.

The report should adhere to the following constraints:

- Limit the report to maximum of 5 pages in total.
- We provided you with an Overleaf template to make your document, and it is easier for collaboration.
- Please adhere to the format (e.g., font size, spacing, ... etc.).
- The template is in view only, so please create your own project and copy-paste the template's `main.tex` to your own report's `main.tex`.
- In this report, include the following items / answer the following questions with

Q1. Architecture drawing and description.

- Provide a clear block diagram of your accelerator, including the building blocks like MAC array, controller, bus, and memory. Clearly show the shape and organization of your GeMM array(s)
- Discuss the key trade-offs that guided your architecture and how you came to this design. Support with qualitative arguments about the area and complexity impact of your choices (e.g., interconnect, control logic).

Q2. Data flow, data layout and address generation.

- Briefly explain how data flows through your architecture for each of the three workloads. This can be illustrated with the nested (parallel/sequential) for-loop representation.

- Explain how matrices A, B, and C are stored in memory and how your address-generation units support this layout.
- Describe how the operational for-loop structure maps to the addresses produced for at least one important memory interface.
- Argue why this layout and loop structure are a good match for your array and dataflow. (Refer to Lecture 3 for the for-loop discussions.)

Q3. Accelerator utilization, bandwidth and performance

- Discuss the latency (in cycles) and spatial utilization of the MACs for each workload
- Quantify the memory bandwidth usage of your current design by counting how many data words enter and leave the accelerator per cycle at steady state.
- Identify the most likely critical path in your datapath at a qualitative level.
- If you needed to shorten the critical path (e.g., after a timing review), what design changes would you make?
- Discuss the cost of these modifications in terms of area, latency, and control complexity.

Q4. Roofline model of your accelerator.

- Draw the roofline model of your accelerator, with your maximum compute throughput (compute roof) and memory bandwidth (memory roof).
- Compute and mark the arithmetic intensity and operating point for each of the three workloads.
- Discuss how well these operating points match the latencies observed in your simulations.

Q5. Up-/down-scaling scenario's.

- Discuss how you would change/scale the design if you were allowed to increase to 256 PEs. Explain how you would adapt the topology and memory system, and discuss the main impact of this scaling in terms of bandwidth, utilization, and implementation complexity. How does your roofline plot change?
- Discuss how you would change/scale the design if you had to constrain your available memory bandwidth to half, or one-quarter of your current bandwidth. Explain how you would adapt the topology and memory system, and discuss the main impact of this scaling in terms of throughput, utilization, and implementation complexity. How does your roofline plot change?

Q6. Test coverage and verification quality.

- Summarize which unit tests and top-level tests you executed.
- Indicate what coverage information you obtained (quantitative/qualitative).
- Identify expected coverage gaps and propose additional tests that would improve your confidence in the design.

Q7. Disclosure of using GenAI

- Please find in the Toledo page the *Use of GenAI Form*.
- Please read through the terms.
- Please fill in the details of the form and indicate where you used GenAI.
- **Please include this form in your submission.**

A Getting started

This appendix helps you set up the project repository, compile and run the simulations, and inspect basic outputs such as waveforms and coverage reports. There are also some useful tricks to help you in your design.

A.1 Repository and tools

Here are the following requirements you need:

- You need access to ESAT computers as the commercial tool Questasim is provided for you.
  - Option 1: It is easier to use one of the computers provided in the computer classrooms (rooms 01.62, 02.54, and 01.54).
  - Option 2: If you prefer to work in your own laptop/computer, please use ESAT virtual desktops. You can find instructions in: `https://it.wiki.esat.kuleuven.be/en/students/virtualdesktops`. IMPORTANT: Use Rocky Linux 8 as there is a problem with Questasim for Rocky Linux 9. Select the SPICE Console after installing virt-viewer.
  - Option 3: Use VNC viewer. You can find instructions in: `https://it.wiki.esat.kuleuven.be/en/working-remote`. Simarly for this option, you will have to use a server with Rocky Linux 8.
- All the tools you need are available on the local machines or the remote machine you tunneled to.
- Clone the repository with:

```
git clone https://github.com/KULeuven-MICAS/cpaep2526_project
```

- The main simulator is Questasim. It is a commercial tool already provided by ESAT.
- Once you have access to your virtual desktop and you have cloned the repository, please the shell script inside of the repo to source Questasim dependencies:

```
source ./questa_setup.sh
```

- Check if Questasim works by running:

```
vsim -help
```

- A help log should appear if sourced correctly.
- Inspect the cloned repository. The directory structure and contents are a simplified version of a typical digital designer's setup.

- `rtl/`: RTL source files for MACs, controllers, memories, etc.
- `flists/`: Filelists which are used for listing the necessary files for simulation.
- `tb/`: Testbenches (unit and top-level).
- `includes/`: Contains header files. Typically used in testbenches for functions, tasks, and sometimes generalized parameters.
- `docs/`: Additional documentation (if any).
- `work/`: (You won't see this at a fresh clone). This work directory contains any temporary files used for compilation and simulation. It is generated after every newly generated test you do.

A.2 Running your first simulation

Upon a successful clone and setup for Questasim, let's see if you can run simulations.

- We have already automated the compile and simulate steps for you. Please invoke the following command to run a simple MAC PE test:

```
make TEST_MODULE=tb_mac_pe questasim-run
```

- You should see the following log:

```
# Loading sv_std.std
# Loading work.tb_mac_pe(fast)
# Loading work.general_mac_pe(fast)
# Test 0 passed.
# Test 1 passed.
# Test 2 passed.
# Test 3 passed.
# Test 4 passed.
# Test 5 passed.
# Test 6 passed.
# Test 7 passed.
# Test 8 passed.
# Test 9 passed.
# All tests passed!
# ** Note: $finish    : tb/tb_mac_pe.sv(145)
```

- You can run the simulation with a GUI interface with:

```
make TEST_MODULE=tb_mac_pe questasim-run-gui
```

- You should be able to see a GUI window similar to Fig. 1.
- Then the same log should be seen on the bottom panel.

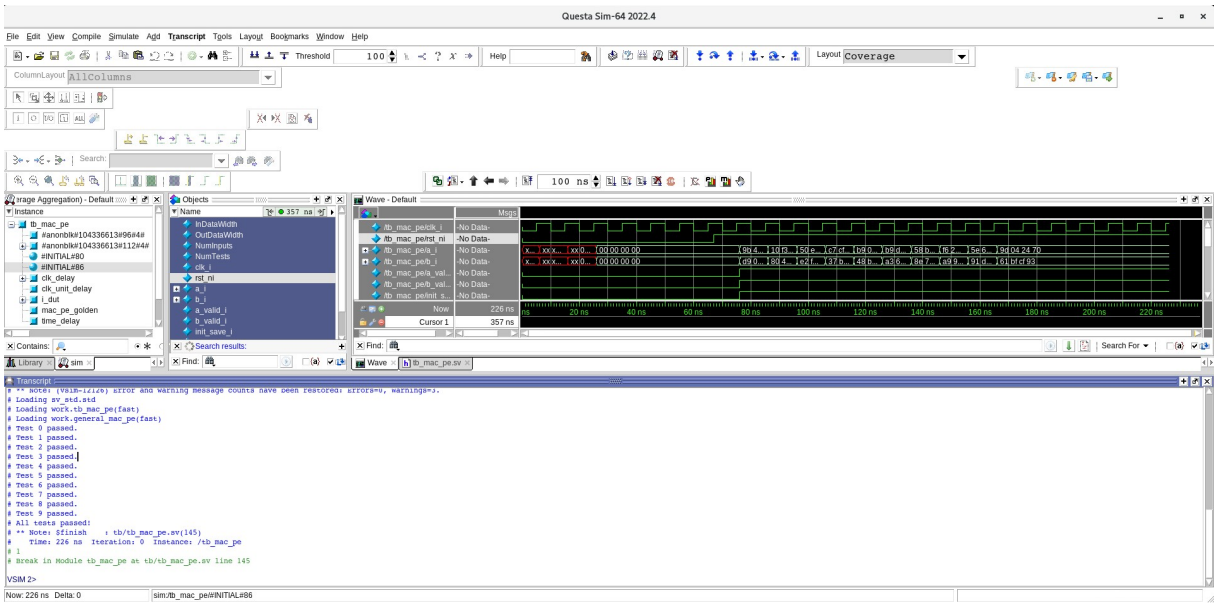

Figure 1: Initial VSIM window

- Verify that you can run the baseline 1-MAC GeMM end-to-end before modifying the design.

```
make TEST_MODULE=tb_one_mac_gemm questasim-run
```

- You can clean the current simulations and dumped files with `make clean`
- If you are interested in the compile and simulation commands, you can look into the `Makefile`.

A.3 Viewing waveforms

When debugging, you will often want to inspect waveforms. Thankfully, Questasim is easy to use for viewing waveforms.

- The waveforms can be viewed by invoking the GUI for Questasim.
- Sometimes the waveforms don't appear by default; however, the default view shows three main panels as shown in Fig. 2.

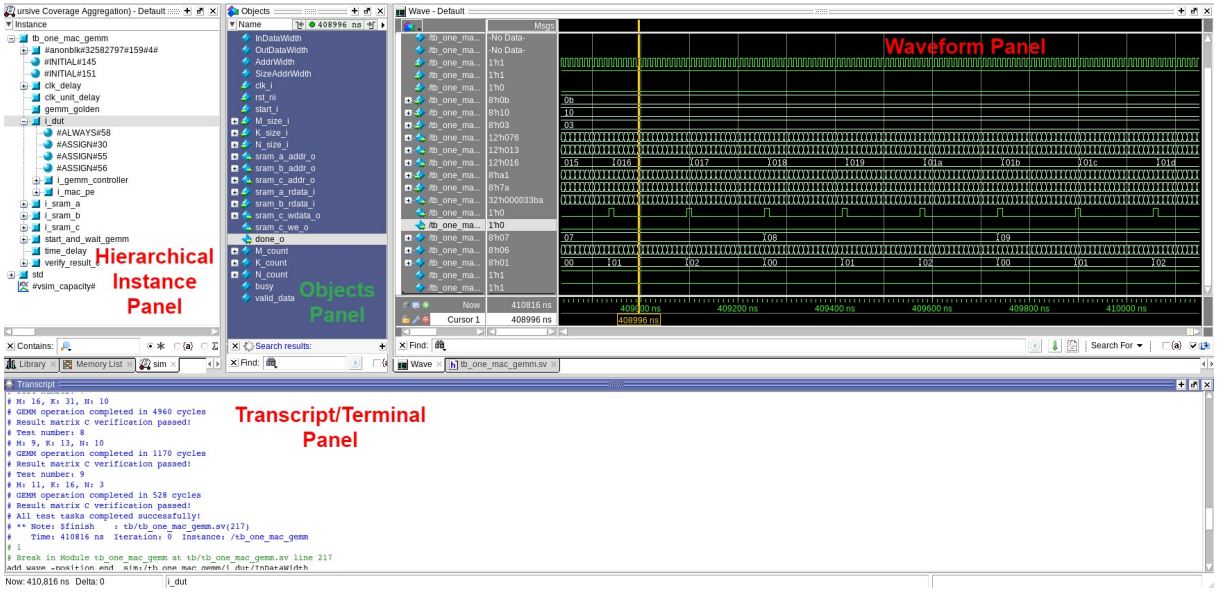

Figure 2: Default Questasim panels

- The left-most panel is the *hierarchical instance panel*, which shows the hierarchy of instances and modules from the testbench down to the lowest sub-block level.
- The more sub-blocks you have the deeper the hierarchy goes.
- In the same instance panel, you can click on the + boxes to drop-down the hierarchy.
- The middle panel is the *object panel*, which shows the objects (e.g., wires, ports, registers, state machines... etc.) for the selected instance from the instance panel.
- These objects can be viewed in the waveform panel, which is on the right-most part of the figure.
- Select all objects (`alt + a`) or select one, then right-click and select `Add Wave`. Alternatively, you can just click and drag the selected object to the waveform panel.
- This will dump all selected objects into the waveform window.
- You can expand the signals by pressing `ctrl + f` in the waveform window. This expands the entire simulation window.
- You can zoom in and out with the scroll bar or the magnifying glass options on the upper left corner, just above the instance window.
- At the bottom is the *transcript or terminal panel* where you can see the logs of your current simulation.

A.4 The 1-MAC GeMM Setup

Each company or group would have its own customized ways of setting up its test benches, but there are common principles that come back in each approach. Let's check how the design and test environment set up for the 1-MAC GeMM. Use this as a guide on how to create your own setup.

- Fig. 3 shows the instance details of the setup for the 1-MAC GeMM along with the instance names, which you could follow along by inspecting the appropriate codes.

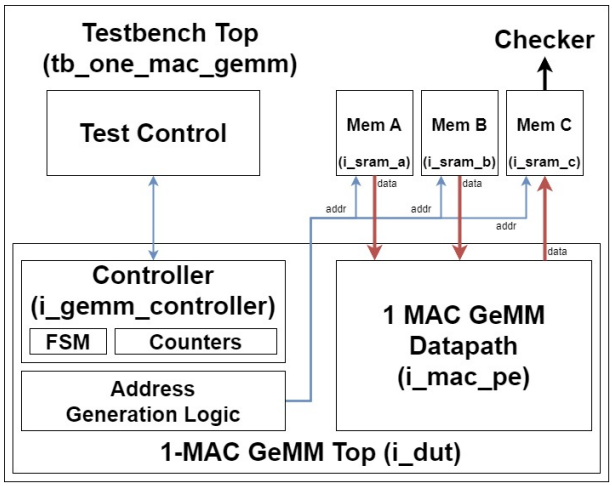

Figure 3: 1-MAC GeMM testbench details

- At the top-most level is the testbench found in `tb/tb_one_mac_gemm.sv`.
- It consists of the 1-MAC GeMM as the *design under test* or DUT (`rtl/gemm/gemm_accelerator_top.sv`), the memories for the inputs and outputs (`rtl/gemm/gemm_accelerator_top.sv`), and the test control signal.
- Inside DUT, there is a 1 MAC GeMM datapath (`rtl/gemm/general_mac_pe.sv`), a controller (`rtl/gemm/gemm_controller.sv`) which consists of counters and the FSM, and an address generation logic which translates the counter values into addresses which go into the memories of the testbench.
- From the testbench code, you can see how it is organized from parameters, to the instantiation of modules, and then by the end, the test signals driver and checker.
- The test signal driver tasks can be found in `includes/test_tasks.svh`, and the golden output checker function can be found in `includes/test_func.svh`. Inspect the content of these files.
- The memory models are in `rtl/common/single_port_memory.sv`. These memories are designed based on foundry's memory macro, and the only changeable part is the memory depth and width.
- For this project, we assume the data is already copied from main memory to these macros beforehand. The accelerator can directly start the computation of the result. Since the memory reads / writes data in row basis, For smaller matrices you can pad zero.
- **Please take time to go through the files listed, as we have placed inline comments to help you understand how the simple 1-MAC GeMM was designed. There are also sections with `DESIGN NOTE` tags, which are suggestions on what to modify.**

- Please also note that these notes are just suggestions or recommendations. **You have the entire freedom on what to modify or create for your accelerator design! Provided you adhere to the specifications listed in subsection 1.2.**

A.5 Simulation Setup

Once you have all the modules you need (e.g., testbench, top module, sub-blocks, ... etc.) you only need to prepare a file list of all blocks you need to run your simulation. We have already prepared a process to make it easier for you.

- Once you have all your files, create a file list in `./flists` directory.
- Inspect `./flists/tb_one_mac_gemm.flist` and see the list of files.
- When you list files, do it in a hierarchical order starting from the bottom. For example, the counters and MAC PE are declared first as they have no sub-block dependency. However, the controller is listed after because it needs to use the counters.
- Make sure the name of your file list will be the same name you used for your testbench. In this case, `tb_one_mac_gemm` is the name used for both the testbench and file list.
- When you invoke the `make TEST_MODULE=tb_one_mac_gemm questasim-run`, the argument `TEST_MODULE` will use the same name to search and compile the file list and test bench altogether.

A.6 Using unit tests

Unit tests help you validate individual modules before integrating the full design.

- The current repository already has a bunch of unit tests, which can be useful for you to use.
- In practice, unit tests are meant to approach a design in a step-by-step manner, and it is well considered a good design practice in industry.
- Unit tests also serve as indirect documentation because they provide information on how a certain module is used.
- Feel free to check how each unit test is built, and it can help you understand how some of the basic blocks are used.
- You are free to create your own unit tests as you see fit; however, it is NOT required to do so (nor will it be checked), but it can help make your design easier.
- Currently, we have the following unit tests:
  - A MAC PE test that tests multiple simultaneous inputs.
  - The counter test, which tests the basic functionality of a counter (This counter is used in the FSM of the controller).
  - The memory test, which you can use to check and understand how the memory is interfaced.

A.7 Basic test coverage

Test coverage information gives a rough sense of how thoroughly your tests exercise the different modes and parts of your design.

- Go and run the 1-MAC GeMM with GUI enabled.
- On the top-bar select `Tools > Coverage Report > HTML`. This will open the *Coverage HTML Report* window.
- In the *Verbosity* box select *Source Code* and *All Coverage Types*. Click OK afterwards.
- This will open an HTML view where you can view the test/verification coverage for the instances in your design. See Fig. 4

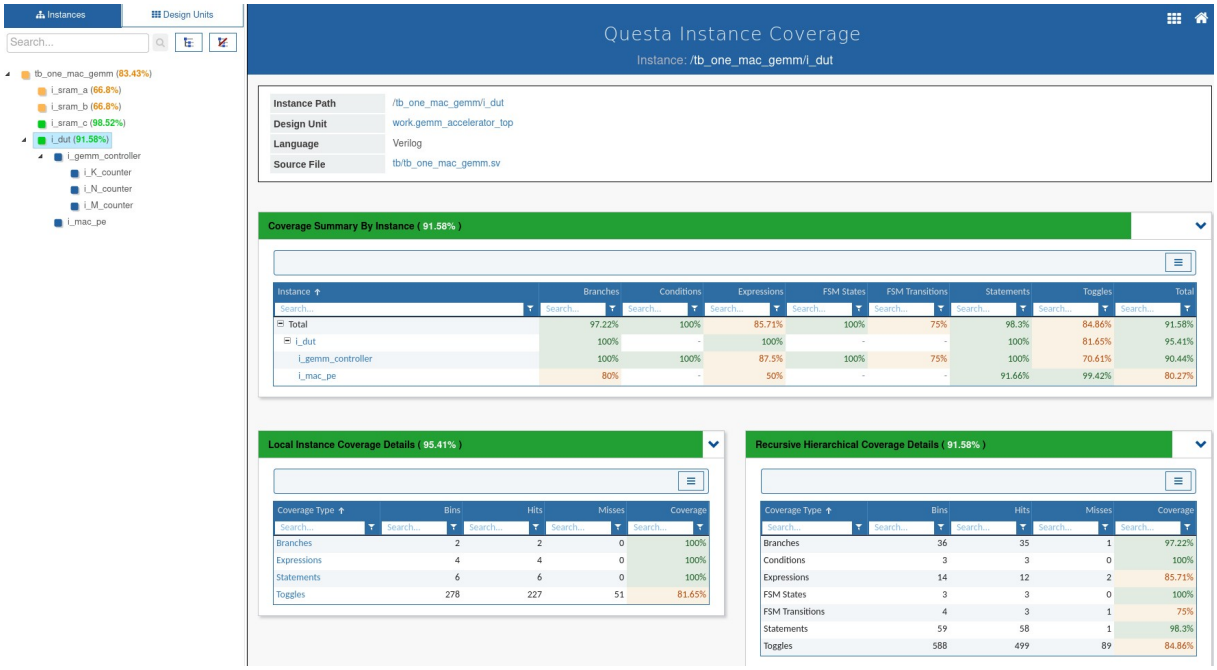

Figure 4: Example of coverage report

- On the left is again the instance view, where you can select which blocks to inspect.
- In the middle is the actual report of the test coverage percentage. Each percentage is a measure of how many lines of the code (depending on the category) were executed or tested in your current setup.
- *Branches*: Whether each decision point (if/else, case items, conditions in loops) hit all possible paths. For example:

```
if (x > 10)
   foo();    // True branch
else
   bar();    // False branch
```

- *Expressions*: refer to the boolean sub-expressions inside a condition were evaluated with all possible truth values. E.g. In the example below, did each `a && b` and even the `|| c` execute?

```
if ((a && b) || c)
```

- *Statements*: whether each executable statement ran at least once during simulation. E.g. Did `a=b` execute? Did `if (x) y = z;` execute? and also did `foo();` execute?

```
a = b;
if (x) y = z;
foo();
```

- *Toggles*: Whether every bit in your nets/regs toggled from 0→1 and 1→0 during simulation. E.g., For example, if every bit of `logic [3:0] a` toggled from 1 to 0 and vice versa at least once.

B Design and implementation guide

This appendix suggests a possible workflow for completing the project. You do not need to follow it strictly, but it is designed to help you move from the baseline design to a well-tested 64-MAC accelerator in a structured way.

Step 1: Understand the baseline

- Read this assignment and the report template.
- The report template provides hints about what to expect from the design.
- Run the 1-MAC GeMM implementation and gain a deeper understanding of the modules and how they work together.
- Read the inline comments in the code, as they point to the parts you will need to modify.
- Run the baseline testbench (see Appendix A) and inspect a waveform for a small case to understand dataflow and control.

Step 2: Sketch possible array shapes

- Sketch multiple ways to arrange 64 MACs (e.g., 8×8, 4×16, hierarchical, or configurable designs).
- For each option, consider how cases 1–3 map to the array (tiling, reuse, idle MACs).
- Make a rough comparison of expected utilization and control complexity before selecting a design.

Step 3: Decide on dataflow, loop structure, and data layout

- Write down the nested loops that describe your GeMM computation, identifying which loops run spatially and which run temporally.
- Choose how A, B, and C are laid out in the given memories (e.g., row-major, column-major, block-based) and how tiles map to memory banks.
- Ensure that your data layout aligns with your array shape and chosen dataflow to avoid unnecessary stalls or bandwidth waste.

Step 4: Extend the datapath to 64 MACs

- Reuse and customize the MAC block to instantiate your 64-MAC array.
- Consider organizing the MAC units hierarchically (e.g., a PE block, a row of PEs, and a top-level block aggregating multiple rows).
- Add the required interconnect and registers to implement your dataflow.
- Simulate small synthetic unit tests (e.g., tiny matrices) to verify correct array wiring before integrating with full memories.

Step 5: Adapt control and address generation

- Modify or replace the main FSM controller so it implements your loop structure and orchestrates the full array.
- Design address-generation units that produce correct access patterns for A, B, and C under your chosen data layout.
- Start with one workload (e.g., case 3) and get it fully working before extending support to all three cases.

Step 6: Integrate with the provided memory system and testbench

- Connect your array, controller, and address generators to the provided memories and the top-level testbench.
- Re-run the provided tests and debug mismatches using waveforms.
- Once one workload passes, extend and test the remaining workloads.

Step 7: Refine and optimize

- Identify the likely critical path (e.g., wide combinational logic between registers) and consider adding pipeline stages if necessary.
- Check that your array maintains good utilization for all three workloads; if not, revisit the mapping or array shape.
- Add targeted tests that stress specific parts of the design (e.g., boundaries, partial tiles).
- Revisit the FSM, address generation logic, and testbench driver to ensure they align well with your chosen design flow.

Step 8: Quantitative evaluation

- Measure the latency (in cycles) for the three workloads using your simulator.
- Count the total number of data bytes entering and leaving the design per cycle to estimate memory bandwidth usage. Reminder: since SRAM reads/writes in rows, even some of the data is unused, these data should still be counted.
- Compute arithmetic intensity for the workloads and build the roofline plot required for the report.
- Explore hypothetical scaling to 256 PEs and bandwidth-constrained scenarios (1/2 and 1/4 bandwidth), as required by the design questions.

Step 9: Verification and coverage

- Ensure that unit tests still pass after your modifications; add tests for new modules (e.g., controllers or array wrappers).

- Run coverage collection and identify parts of the design that are poorly exercised.
- Add additional tests to improve coverage where it is clearly lacking.

Step 10: Finalize the report

- Use your measurements and observations to answer the design questions in Section 2.
- Include clear figures (e.g., architecture diagrams, simplified timing sketches, roofline plots) where beneficial.
- Check that your explanations match the behavior observed in the simulation and that your trade-off arguments are clearly articulated.