CPAED Exercise Sessions 2021-2022

Nimish Shah, Koen Goetschalckx, Kodai Ueyoshi,
Man Shi, Marian Verhelst

1 Introduction

In this project, you will work on a design optimization for a neural network (NN) accelerator by implementing/modifying one yourself. Moreover, you will also be introduced to a typical SystemVerilog testbench set-up and familiarize with it by adapting and completing it to properly verify your own NN accelerator. Section 2 recaps convolutions as seen in convolutional layers of CNNs. The task for this exercise sessions series is given in section 3. Section 4 gives an overview of a simple accelerator, which contains just a single MAC-unit, and an accompanying testbench, both given to you as a baseline. Some practicalities related to the work flow (e.g. how to run a simulation) are addressed in section 5. Please read this document completely and carefully before the first session.

2 Convolutional layer recap

Built upon the normal, well known, plain 2d convolution, where a window of weights sweeps over all possible locations in an input image, element-wise multiplies the weights with the pixels in the window and then adds up the products to generate a pixel in the output, the convolution in convolutional layers of a CNN is an extension on this. The extension adds an extra dimension to the input and, correspondingly, to the weights. This dimension is also summed over after the element-wise multiplications, just like the original 2 dimensions were. Next, an extra dimension is also added to the output. This is done by repeating the whole process for different sets of weights, each one generating a separate 2d output. These are then stacked to get the 3d output. See Figure 1.

3 Task

The main task of this project is to make your own design of a NN accelerator for a workload consisting of a single convolution (CONV) layer (see 3.3), and verify it properly. This boils down to:

1. **Write/modify RTL code** for the overall system. If you want, you can start from the provided code.

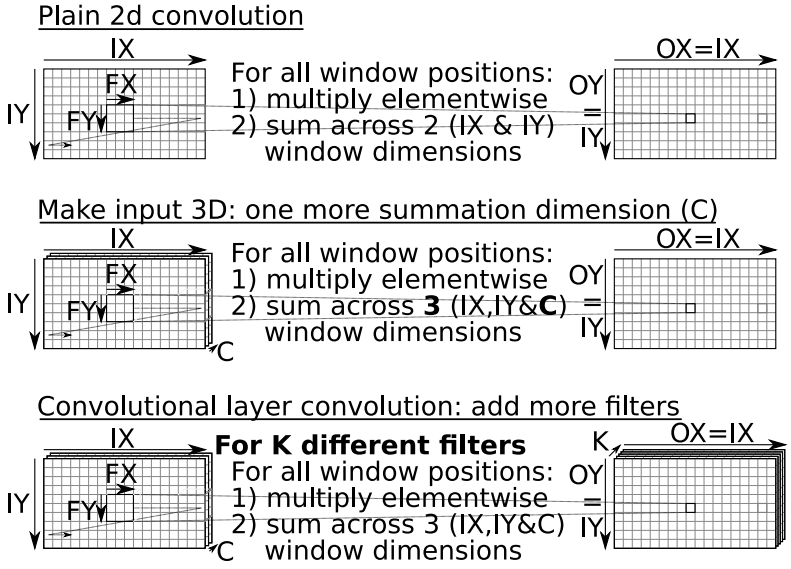

Figure 1: Convolution layer, regarded as extension of plain 2d convolution

2. **Create/update the testbench**.

3. **Optimize the design** targeting minimal product of latency and energy under given area and testbench bandwidth constraints.

4. **Test and verify** your design, reporting the achieved performances and test coverage. Add custom covergroups and coverpoints for functional coverage wherever necessary.

3.1 Delivery and pairing

Deliver your solution and a short report by **Nov 22th 24h00**!

For the solution, include all RTL and testbench code, i.e. the `src` directory. In the report, 1) **clarify the design choices** you made and the **reasoning** behind them (e.g. MAC-array dimensions, temporal and spatial loop unrolling and loop order, memory hierarchy, etc.) and 2) **provide the latency, energy, area and IO bandwidth** of your obtained solution. There is no need to for instance re-iterate the testbench structure already given in this document. However, 3) **specific noteworthy changes to both DUT and testbench** should be included. Finally, 4) report the **test coverage** results.

Bullet-point style is allowed and encouraged. The report should not be longer than 2 sides of an A4 (with fontsizes ≥ 11 and some page margin).

If you want, pair with a classmate, as this project can be executed in groups of 2. Jointly sign up for one of the Toledo groups, and split the tasks.

One joint report + solution submission per pair. However, **both partners should be fully aware** how everything works. In other words: *"My partner did that part"* is not an excuse not to know an answer on the oral defence during the exam.

3.2 Some competition

To give you a better feel of how good your solution is compared to others, please upload your best results to this form:

`https://forms.office.com/r/dCsxypwVWx`

You can view the results at:

`https://kuleuven-my.sharepoint.com/:x:/g/personal/kodai_ueyoshi_kuleuven_be/ER55kho73IZGoUymMOpw_OQB057s6Md0DRx9CfAshxWu7g?e=Zf4uS1`

3.3 Workload

The benchmarking CONV layer is given in Table 1. Figure 2 illustrates the CONV layer configuration. These dimensions can be considered fixed. Optimizing towards these specific dimensions is thus allowed and encouraged. We do not ask for an accelerator that has high performance over a broader range of CONV layer dimensions.

Table 1: CONV benchmark parameters.

	Parameter	Number		Parameter	Number
Input width	IX	64	Output width	OX	64
Input height	IY	64	Output height	OY	64
Input channels	C	4	Output channels	K	32
Kernel width	FX	3	Stride	S	1
Kernel height	FY	3	Bitwidth(A&W)	-	16 bits

3.4 Target & constraints

3.4.1 Target

The goal is to **minimize the product of latency and energy** usage.

3.4.1.1 Subtarget: latency

If you use the provided registers, multiplier, adder, fifo, and memory modules where applicable (instead of for instance using `*` directly), **latency is reported by the simulation**. You can **change the clock frequency** at the top of the testbench (integer values only). If you make it too short for your critical path (see below), you will get errors.

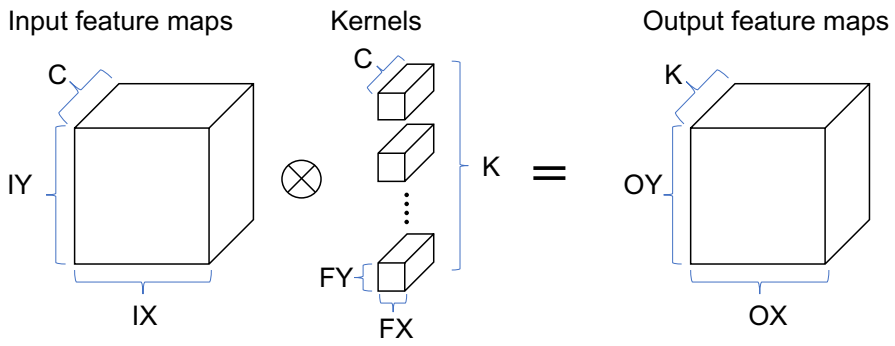

Figure 2: Convolution layer.

For insight and quick estimates, latency can be approximated as the (required **number of clock cycles** to run the CONV layer) **multiplied** by (the time necessary to complete **one clock cycle**).

$$latency = \#cycles \cdot duration/cycle$$

The latter factor is equal to the *critical path length*. For this project, we approximate this as the highest number of 16b multipliers and/or 32b adders on any single path between two registers or inputs/outputs of the chip. Figure 3 shows some examples.

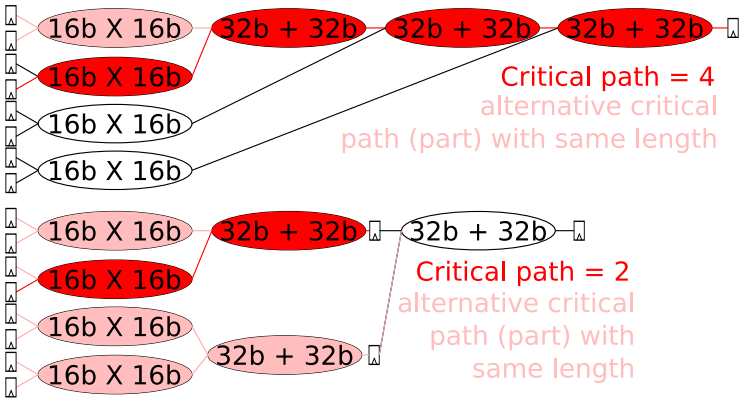

Figure 3: Critical path examples

The $\#cycles$ can approximately be factorized as follows:

$$\#cycles \approx \frac{\#macs\_to\_be\_done}{\#average\_number\_of\_macs\_done\_per\_cycle}$$

This can further be factorized into:

$$\#cycles \approx \frac{\#macs\_to\_be\_done}{\#mac\_units\_in\_design \cdot their\_utilization}$$

Thus, **to decrease latency**, one can

1. increase the **number of mac units** in the design, or
2. increase their **utilization**, or
3. decrease the **critical path** length

without equally harming the other two factors when doing so.

However, you can not infinitely enhance these factors, as you are limited by the **constraints** below.

3.4.1.2 Subtarget: Energy

Energy are composed of two parts: MAC energy and memory access energy. The former one is constant for a given workload, and determined by:

$$MACenergy = \#mac \cdot unit\_mac\_energy$$

The **memory access energy** is determined by:

$$\begin{aligned} Memory\_energy =& (\#SRAM\_bits\_transferred \cdot SRAM\_bit\_energy) \\ &+ (\#Ext\_Mem\_bits\_transferred \cdot Ext\_Mem\_bit\_energy) \end{aligned}$$

The energy can be calculated by detecting the read/write enable signals of memories and looking at the valid and ready signals from testbench to DUT interface. The related energy costs of operations are listed in Table 2.

Table 2: Energy relative costs.

Module name	Energy relative cost
External memory/FIFO or testbench(*)	1/data bit read or written
On-chip memory/FIFO (SRAM)	0.1/data bit read or written
(16b x 16b + 32b) MAC operation	0.001

If you use the provided memory and FIFO blocks for your memories, the energy cost of these is calculated automatically and reported by the simulation. Regarding the bandwidth between the testbench and DUT, have a look at the bottom of `intf.sv`. **Adjust this code when you change the interface** between DUT and testbench to keep the automatic energy calculations correct.

3.4.2 Constraint: Area

The relative area cost to be used is shown in table 3. The total area that can be used is limited to **500,000**. External memories (e.g. **ext_mem** in the example below) are excluded from this cost. However, communicating with these memories has a higher energy cost (see above). **If you use the provided registers, multiplier, adder, fifo, and memory modules** where applicable (instead of for instance using * directly), **area is reported by the simulation**.

Table 3: Area relative costs.

Module name	Area relative cost
External memory/FIFO	0
On-chip memory/FIFO, 256 words or more	1/bit
Flipflops or on-chip memories/FIFOs less than 256 words	17/bit
Multiplier (16bx16b->32b)	5800
Adder (32b+32b->32b)	1000

3.4.3 Constraint: Bandwidth

The **bandwidth between the testbench and the DUT is constrained to 48 bits.** You are not allowed to average this: you can have at most 48 signals connecting the DUT to the testbench **(*)**. It is thus not allowed to have for instance 96 and say that the bandwidth is only 48 because you use them only every second cycle. If you want to do something like that, receive 2 times 48 in consecutive cycles and repack the data in the DUT. Within the 48 bits you are free to decide how many of these are inputs, outputs or half-duplex (bidirectional). **(*):** handshaking signals may be excluded from this constraint. `output_x/y/ch` may be excluded as well, as we included these only to make the verification of the output independent of dataflow and could be removed making the monitor aware of the dataflow.

4 Provided example files

Code for **a baseline accelerator and testbench is provided** to you as a starting point and inspiration. You can find the example code of the DUT in the `device` folder, of the RTL module components in the `rtl_building_block` folder, and of the testbenches in the `test` folder. The provided testbench and DUT (**top_system**) can be run and simulated without modifications. **Smartly copy-pasting and adapting the given code is allowed and encouraged!**

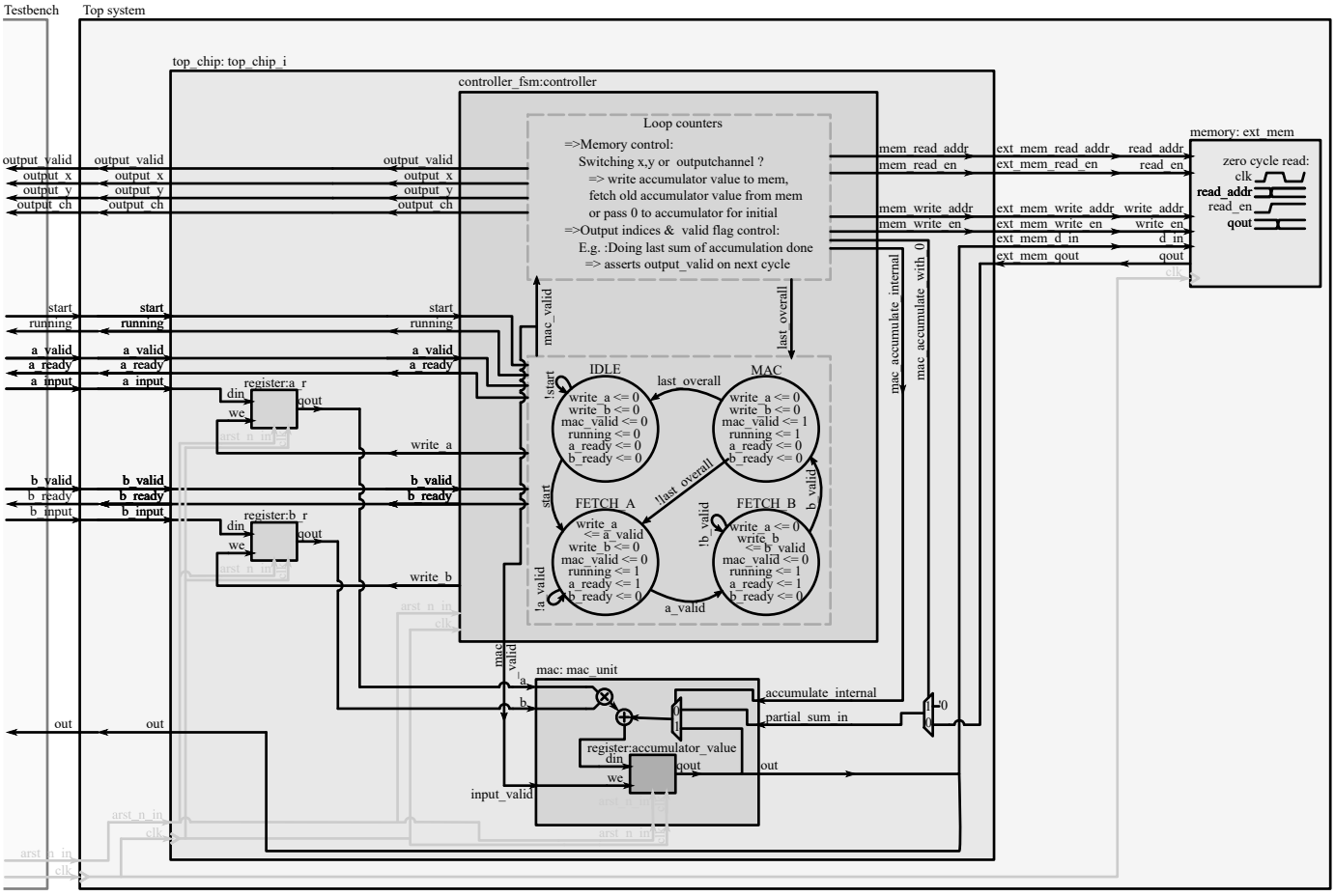

Figure 4: Device Under Test overview

4.1 Device Under Test: a single MAC-unit, a controller, and an external memory

The **DUT**, shown in Figure 4, exists of two main parts: the **top_chip** and **ext_mem** memory.

As a starting point, the provided **top_chip** consist of single **MAC unit**. Its data is driven by the registers holding **a** and **b**, as well as the **ext_mem** when temporary results are reloaded.

All of this is controlled by the **controller**. This consist firstly of a 4-state FSM looping over *fetch a* from testbench, *fetch b* from testbench, and use these fetched inputs to do a *MAC*. Secondly, there are **loop counters** corresponding to the CONV-kernel `for` loops. By changing when the counters increase and reset, the order of the `for` loops, a.k.a. the **dataflow**, can be altered. As a rule

of thumb, the counter of an outer `for` loop increases when all inner `for` loop counters are reset. The innermost loop counter (as provided) is incremented when the small FSM indicates a MAC operation is happening.

Based on the current indices in these loops, as given by the loop counters,

1. the **ext_mem** address and control is generated, as well as
2. the **output valid** and indices signals.
3. the FSM is signaled when the overall last MAC is being done, after which the FSM goes back to the **idle** state until a new *start*.

Note that you are free to change the dataflow (and thus loop order) as you see fit, free to introduce on-chip memories to cache data values, free to expand the current MAC-'array' to have multiple MAC-units, etc. as long as your design meets the given bandwidth and area constraints.

Handshaking in the example is implemented with `valid` (or `vld`) and `ready` (`or rdy`) signals. These indicate that data on a line is valid and that the receiver is ready to take it, respectively. **Data is therefore passed if and only if both `valid` and `ready` are high**. If one of them is not present as a part of an interface, it is assumed to be high by the other side. For instance: the DUT assumes the testbench will immediately consume any data indicated by `output_valid`. Note that it is possible to use no handshaking signals at all. In this case, both the sender and receiver of data must work on the same predetermined and hardcoded cycle accurate schedule of when data is passed. Using handshaking is usually easier as you can change only one side without harming functionality. If you do make changes to the handshaking, adjust the energy calculation code at the bottom of `intf.sv` correctly.

4.2 Testbench

A typical SystemVerilog testbench, shown in Figure 5, consists of:

- An **interface**: describes the interface between the DUT and the TB.
- **transaction** classes: groups variables that belong together. TB components communicate by passing transactions via mailboxes
- A **generator**: generates test data (as transactions). In this case, this is a kernel and a feature map. This data is passed to the **driver**, which sends it to the DUT, and to the **checker**, which calculates the expected outputs based on it.
- A **driver**: consumes transactions (logical view) from the generator and accordingly drives DUT inputs (physical view) through the interface. In this case, the driver sends the features and weights to the DUT in the order it expects it. This order is determined by the **dataflow**, implemented by the loop counters (see above) in the DUT controller. **Thus, when changing the loop order (dataflow), equivalent changes should**

be made to driver and the loop counter logic in the controller fsm, so that the order of sending data and receiving data is kept equal to each other.

- A **monitor**: monitors the DUT, especially its outputs, through the interface, and creates corresponding transactions to package these outputs. These transactions are then sent to the **checker**, that compares them against the expected outputs.
- A **checker**: checks the outputs of the DUT, received as a transaction from the **monitor** for correctness by comparing the results in the transaction a golden reference function applied to the inputs received from the **generator**. It sends its findings on correctness to the **scoreboard**.
- A **scoreboard**: keeps track of the DUT's score (eg. success percentage). Also controls the **number of tests** to be done before **finishing the whole simulation**.
- An **environment**: instantiates, connects, and launch the execution of all the above testbench components.
- A **testprogram**: instantiates and runs the environment.
- The **Device Under Test**, see above for contents.
- The **testbench** itself: instantiates the testprogram, the DUT, and the interface to it.

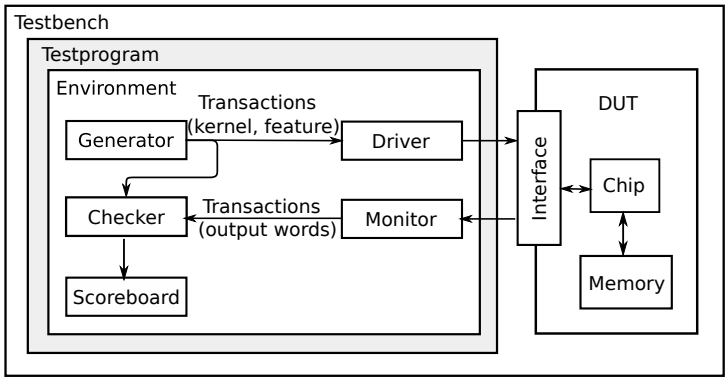

Figure 5: Testbench overview. Driver, monitor and interface need to be modified in this session.

5 Practicalities

5.1 Startup

Once graphically logged in on Linux, and open a terminal (Applications, System Tools, Terminal). In the terminal, run:

```
cd ~/Documents
mkdir -p CPAED
cp -R /users/micas/kgoetsch/Documents/CPAED/to_students/21221 CPAED
cd CPAED/21221
```

(you can paste commands with `ctrl+shift+v`). This makes a 'CPAED' folder in your Documents folder, makes a copy the necessary directory for this exercise session, called '21221', into it, and finally goes into this '21221' folder. Keep the terminal open.

5.2 Simulator compilation/launching/... commands

After above subsection you can run the following commands in that terminal:

- `make clean`: clean automatically generated files. Useful when things don't seem to work anymore.
- `make compile`: compiles your hdl code (**see Note 2!**) to an executable simulator.
- `make run`: runs that executable simulator. Do not forget to execute `make compiles` first when you made changes to the hdl code.
- `make all`: shorthand for `make compile` and then `make run`.
- `make compile_gui`: compiles your hdl code (**see Note 2!**) to an executable simulator with support for the Graphical User Interface, which includes a wave-viewer, line-by-line debugger, etc.
- `make run_gui`: runs that executable GUI simulator (**see Hint!**).
- `make gui`: shorthand for `make compile_gui` and then `make run_gui`.
- `make coverage`: compiles and runs the testbench, with coverage options enabled.
- `make coverage_show`: after `make coverage`, opens the generated coverage report in a browser.
- `make browsesource`: opens the filebrowser on the directory with source code.

Thus, the **default work flow** is: edit hdl code and then run `make all` for a quick test without GUI, or `make gui` for debugging with GUI (see below section; see Hint below).

Note: these `make` commands only work when your terminal is in the folder where the makefile is in.

Note 2: only the files listed in `src/sourcefile_order` are compiled, and in that order. Make sure this file is updated correctly whenever you create a new file or dependencies change!

Hint: when the GUI is already opened, don't run `make run_gui` or `make gui`. Instead, only run `make compile_gui` and then enter `restart` in the command window of the GUI itself.

5.3 Simulator (VCS) GUI (DVE)

After starting the gui (with `make run_gui`, see above), you get a window that looks like Figure 6.

After creating a wave view (through the shown right-click menu), you get a wave-view window like Figure 7.

The usual GUI workflow is:

1. Edit source code
2. Call `make compile_gui` to update the simulator to your adapted source code
3. Either start a gui with `make run_gui` or restart the existing one with the restart button (or by typing `restart` on its command line)
4. Run the gui by clicking the run button or entering `run` (possible followed by a time, e.g. `run 1ns`) in the command line
5. Inspect the waves
6. Realize you've added to few signals to the wave view, so add more signals
7. Restart simulator and run again
8. ...
9. Understand what's going on
10. Repeat

5.4 Online references for System Verilog

1. `https://www.chipverify.com/systemverilog/systemverilog-tutorial`
2. `https://www.doulos.com/knowhow/systemverilog/systemverilog-tutorials/`
3. `https://www.asic-world.com/systemverilog/tutorial.html`

4. Online playground with examples:
   `https://www.edaplayground.com/`

5. Official SV Language Reference Manual for the interested students:
   `https://ieeexplore.ieee.org/document/8299595`

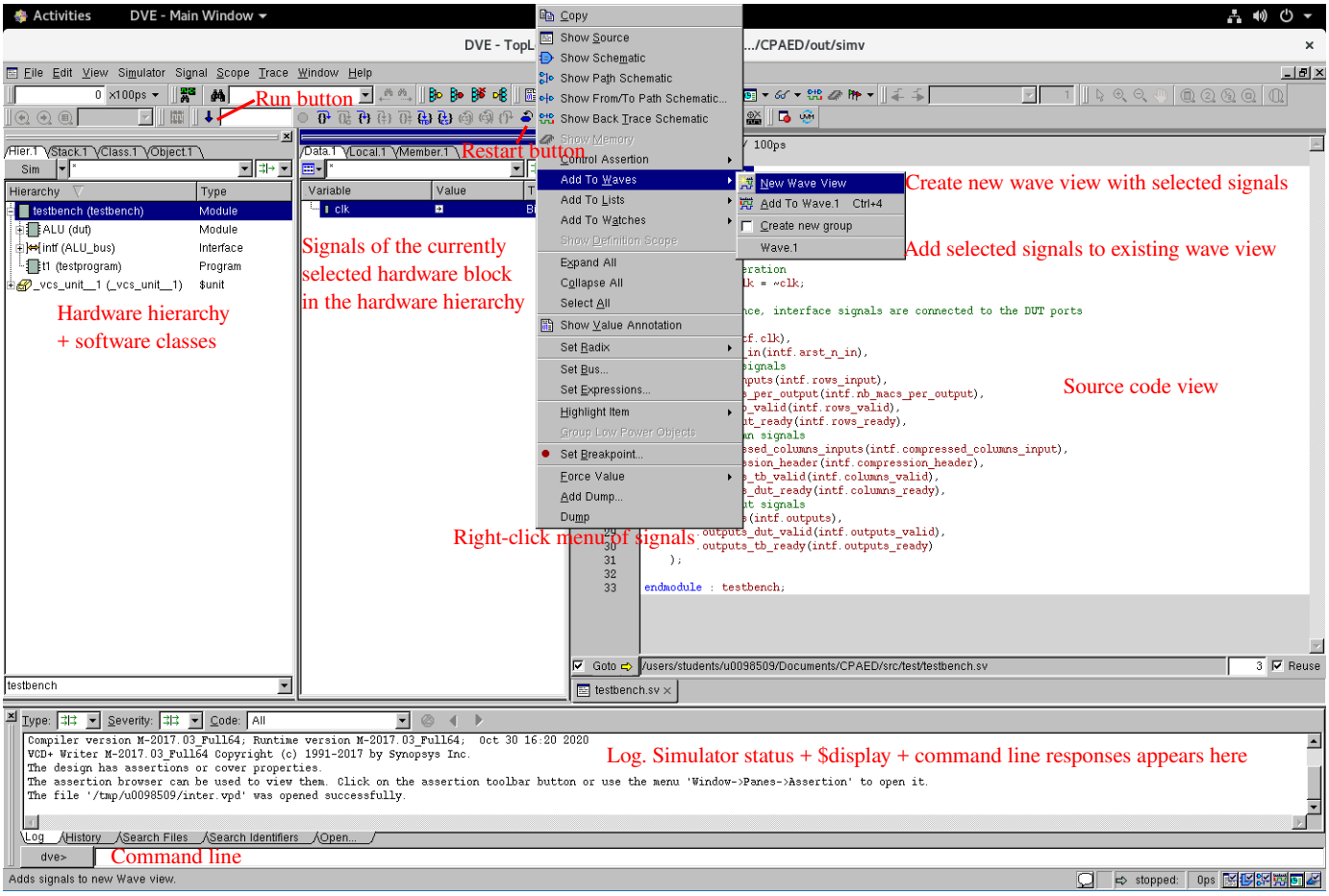

Figure 6: VCS GUI main window

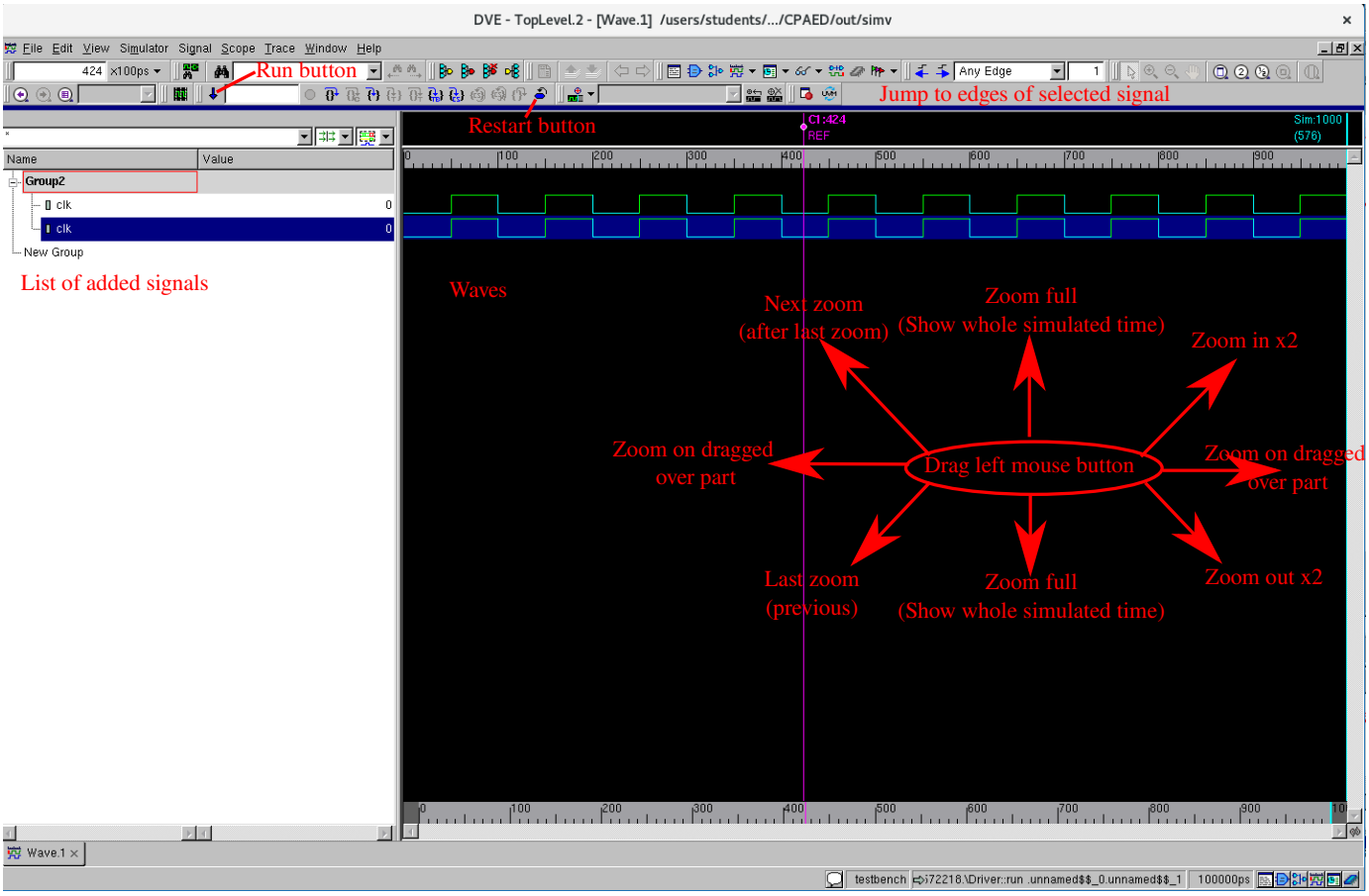

Figure 7: VCS GUI wave view window