CS425: Computer Systems Architecture

Simulation Assignment 1
Assignment Date: 21/11/2025
Due Date: 03/12/2025 - 23:59

In-order processors and code optimization

The purpose of this assignment is to introduce you in simulation and help you familiarize with the details of CPU microarchitecture and code optimization. You have to tune the microarchitecture of an in-order processor, run simple code on it and optimize performance. You will also see the impact of alternative design choices in performance metrics.

For the simulation you will use the gem5 simulator (www.gem5.org). Although the assignment does not require prior experience with gem5, it is recommended that you take some time to follow the gem5 tutorial:

www.gem5.org/documentation/learning_gem5/introduction/

Simulation Assignment 1 - Code Repository

For the infrastructure of this assignment, we have created personal git repositories on the Department's gitlab server for each student enrolled in the course. Your personal repository is in the form:

https://gitlab-csd.datacenter.uoc.gr/hy425_2025f/sim1_submits/sim1-csdXYZWA

Get a clone of your repository with `git clone` using your username (csdXYZWA@csd.uoc.gr). Access is only available via the University network and the VPN. *(Do not fork the repo – Work directly on the cloned git repo)*.

The repo includes the following:

- `README.md` : Quick installation and run instructions
- `code/` : Simple code and Makefile
- `config/` : gem5 configuration of an in-order core based on the MinorCPU model
- `utils/` : Some helpful python scripts for visualization
- `setup.sh` : Shell script for setting environment variables and PATHs
- `run.sh` : Shell scripts for quickly running a gem5 simulation with the given configuration
- `view.sh` : Shell script for visualizing the pipeline behavior in the a terminal.

Study the folders and the files to get an idea of the repo and feel free to create your own scripts as you see fit.

In this assignment you will compile gem5 to target the **RISC-V ISA**. The build requirements can be found here: https://www.gem5.org/documentation/general_docs/building

Check the **`README.md`** file which provides a quick installation and running guide. The gem5 compilation step takes time so complete it as soon as possible!

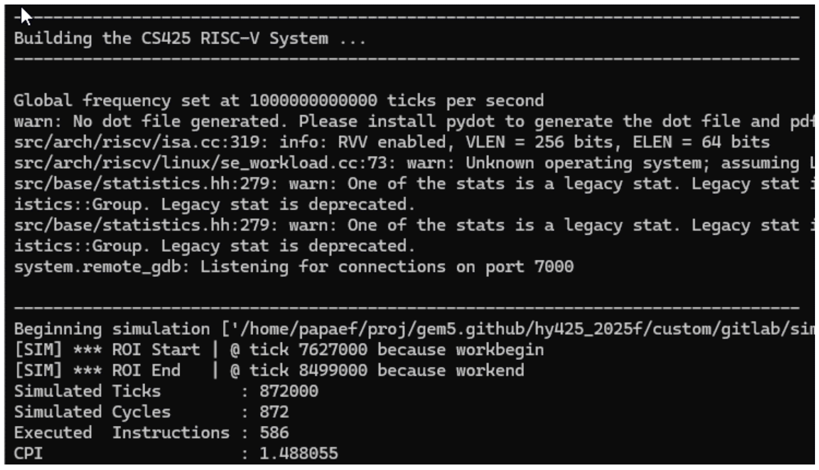
Assignment Tasks

First read the details of the MinorCPU model to understand the architecture that is documented here: https://www.gem5.org/documentation/general_docs/cpu_models/minor_cpu

Now you need to complete multiple discrete tasks for this assignment as follows:

Task 1:

- Check the source code of the `simple_for.c` found in the folder `code`. The code includes Region-of-Interest (ROI) begin and end markers for the important part of the code that you will analyze in this assignment.
- Compile the code using the Makefile. The Makefile also generates a `.dump` file with the disassembled RISC-V instructions. Find the ROI part and have a look at the instructions.
- Run the simulation from the top folder by executing (don't forget to `source setup.sh`): `./run.sh task1`
  The script runs gem5 with the default config file found in (`config/cs425_minor.py`) and places the output of the simulator on a folder called `task1`.
  You should see an output like this:



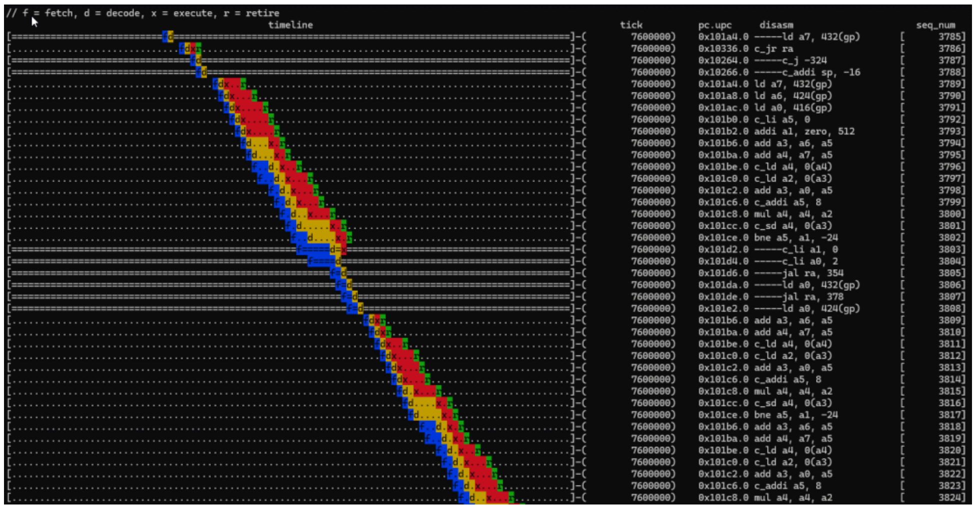
```
--------------------------------------------------------------------------------
Building the CS425 RISC-V System ...
--------------------------------------------------------------------------------

Global frequency set at 1000000000000 ticks per second
warn: No dot file generated. Please install pydot to generate the dot file and pdf
src/arch/riscv/isa.cc:319: info: RVV enabled, VLEN = 256 bits, ELEN = 64 bits
src/arch/riscv/linux/se_workload.cc:73: warn: Unknown operating system; assuming L
src/base/statistics.hh:279: warn: One of the stats is a legacy stat. Legacy stat i
istics::Group. Legacy stat is deprecated.
src/base/statistics.hh:279: warn: One of the stats is a legacy stat. Legacy stat i
istics::Group. Legacy stat is deprecated.
system.remote_gdb: Listening for connections on port 7000

--------------------------------------------------------------------------------
Beginning simulation ['/home/papaef/proj/gem5.github/hy425_2025f/custom/gitlab/si
[SIM] *** ROI Start | @ tick 7627000 because workbegin
[SIM] *** ROI End   | @ tick 8499000 because workend
Simulated Ticks        : 872000
Simulated Cycles       : 872
Executed  Instructions : 586
CPI                    : 1.488055
```

  The output reports the start tick of the ROI and the end tick of the ROI (1 cycle = 1000 ticks) along with the number of cycles, instructions and the CPI. There are also many detailed statistics in `task1/stats.txt`

- View the pipeline behavior of the ROI in the terminal (suggested terminal width >240 columns) by executing:
  `./view.sh task1 7627000`
  You should see an output like this (enter space for more…):



```
// f = fetch, d = decode, x = execute, r = retire
                         timeline                              tick        pc.upc    disasm                          seq_num
                                                        ]-(   7600000)   0x101a4.0 ------ld a7, 432(gp)      [      3785]
                                                        ]-(   7600000)   0x10336.0 c_jr ra                   [      3786]
                                                        ]-(   7600000)   0x10264.0 ------c_j -324            [      3787]
                                                        ]-(   7600000)   0x10266.0 ------c_addi sp, -16      [      3788]
                                                        ]-(   7600000)   0x101a4.0 ld a7, 432(gp)            [      3789]
                                                        ]-(   7600000)   0x101a8.0 ld a6, 424(gp)            [      3790]
                                                        ]-(   7600000)   0x101ac.0 ld a0, 416(gp)            [      3791]
                                                        ]-(   7600000)   0x101b0.0 c_li a5, 0                [      3792]
                                                        ]-(   7600000)   0x101b2.0 addi a1, zero, 512        [      3793]
                                                        ]-(   7600000)   0x101b6.0 add a3, a6, a5            [      3794]
                                                        ]-(   7600000)   0x101ba.0 add a4, a7, a5            [      3795]
                                                        ]-(   7600000)   0x101be.0 c_ld a4, 0(a4)            [      3796]
                                                        ]-(   7600000)   0x101c0.0 c_ld a2, 0(a3)            [      3797]
                                                        ]-(   7600000)   0x101c2.0 add a3, a0, a5            [      3798]
                                                        ]-(   7600000)   0x101c6.0 c_addi a5, 8              [      3799]
                                                        ]-(   7600000)   0x101c8.0 mul a4, a4, a2            [      3800]
                                                        ]-(   7600000)   0x101cc.0 c_sd a4, 0(a3)            [      3801]
                                                        ]-(   7600000)   0x101ce.0 bne a5, a1, -24           [      3802]
                                                        ]-(   7600000)   0x101d2.0 ------c_li a1, 0          [      3803]
                                                        ]-(   7600000)   0x101d4.0 ------c_li a0, 2          [      3804]
                                                        ]-(   7600000)   0x101d6.0 ------jal ra, 354         [      3805]
                                                        ]-(   7600000)   0x101da.0 ------ld a0, 432(gp)      [      3806]
                                                        ]-(   7600000)   0x101de.0 ------jal ra, 378         [      3807]
                                                        ]-(   7600000)   0x101e2.0 ------ld a0, 424(gp)      [      3808]
                                                        ]-(   7600000)   0x101b6.0 add a3, a6, a5            [      3809]
                                                        ]-(   7600000)   0x101ba.0 add a4, a7, a5            [      3810]
                                                        ]-(   7600000)   0x101be.0 c_ld a4, 0(a4)            [      3811]
                                                        ]-(   7600000)   0x101c0.0 c_ld a2, 0(a3)            [      3812]
                                                        ]-(   7600000)   0x101c2.0 add a3, a0, a5            [      3813]
                                                        ]-(   7600000)   0x101c6.0 c_addi a5, 8              [      3814]
                                                        ]-(   7600000)   0x101c8.0 mul a4, a4, a2            [      3815]
                                                        ]-(   7600000)   0x101cc.0 c_sd a4, 0(a3)            [      3816]
                                                        ]-(   7600000)   0x101ce.0 bne a5, a1, -24           [      3817]
                                                        ]-(   7600000)   0x101b6.0 add a3, a6, a5            [      3818]
                                                        ]-(   7600000)   0x101ba.0 add a4, a7, a5            [      3819]
                                                        ]-(   7600000)   0x101be.0 c_ld a4, 0(a4)            [      3820]
                                                        ]-(   7600000)   0x101c0.0 c_ld a2, 0(a3)            [      3821]
                                                        ]-(   7600000)   0x101c2.0 add a3, a0, a5            [      3822]
                                                        ]-(   7600000)   0x101c6.0 c_addi a5, 8              [      3823]
                                                        ]-(   7600000)   0x101c8.0 mul a4, a4, a2            [      3824]
```

  Each pipeline stage has a color and a letter as the legend suggests. Each line draws 100 cycles of execution starting at the tick reported on the right, followed by the instruction address/program counter (PC) and the disassembled instruction. So, the first **f** is at tick 7627000 (cycle 7627). Dot means staying in the same stage/state and equal sign (=) means that the instruction was canceled/squashed at some stage typically due to branch misprediction.

- Report the performance numbers (cycles, instructions, CPI) and your observations in the text file called `answers.md` in the section called Task 1.

Task 2:

- The functional units are not pipelined and there is a simulation flag to model them as fully-pipelined. Run the simulation with pipelined functional units by executing:
  `./run.sh task2 --pipelined-fu`
- View the pipeline behavior and report the performance numbers (cycles, instructions, CPI) and your observations on the performance difference in the text file called `answers.md` in the section called Task 2.

Task 3:

- Modify the Makefile to enable more compiler optimizations like loop unrolling and recompile the `simple_for.c` code and study the `simple_for.dump`.
- Run the simulation with unpipelined functional units (`./run.sh task3_1`) and with pipelined functional units (`./run.sh task3_2 --pipelined-fu`).

- View the pipeline behavior, report the performance numbers (cycles, instructions, CPI), and your discuss your observations on the code and the performance difference compared to the previous tasks in the text file called `answers.md` in the corresponding section.

Task 4:

- Copy the `simple_for.c` to `simple_for2.c` and modify it to instruct the compiler to do loop unrolling using `#pragma` directives (modify also the Makefile to create a new binary `simple_for2` and do not use the `-funroll-loops` flag). Compile it and study the `simple_for2.dump`
- Experiment with different unrolling factors and run the simulation with unpipelined functional units (`./run.sh task4_1 ./code/simple_for2`) and pipelined functional units (`./run.sh task4_2 --pipelined-fu ./code/simple_for2`).
- View the pipeline behavior, report the performance numbers (cycles, instructions, CPI), and discuss your observations on the code and the performance difference compared to the previous tasks in the text file called `answers.md` in the corresponding section.

Task 5:

- Copy the `simple_for.c` to `simple_for3.c` and modify it to do manual loop unrolling using temporary variables like (tmp1, tmp2) and use the macro called `COMPILER_BARRIER()` to control a little bit the order of instructions (modify also the Makefile to create a new binary `simple_for3` and do not use the `#pragma` or the `-funroll-loops` flag). Compile it again and study the `simple_for3.dump`
- Experiment with different unrolling orderings and run the simulation with unpipelined functional units (`./run.sh task5_1 ./code/simple_for3`) and pipelined functional units (`./run.sh task5_2 --pipelined-fu ./code/simple_for3`).
- View the pipeline behavior, report the performance numbers (cycles, instructions, CPI), and discuss your observations on the code and the performance difference compared to the previous tasks in the text file called `answers.md` in the corresponding section.

Task 6:

- Now we will modify the microarchitecture of the CPU by changing the `config/cs425_minor6.py` configuration of the CPU found in lines 80-112. Study carefully and adapt the <u>commit widths</u> to allow for two instructions to complete per cycle.
- Run with `run6.sh` all three versions of the `simple_for` code with unpipelined functional units (`task6_1, task6_2, task6_3`) and with pipelined functional units (`task6_4, task6_5, task6_6`).

- View the pipeline behavior, report the performance numbers (cycles, instructions, CPI), and discuss your observations on the dual-commit pipeline performance compared to the previous tasks in the text file called `answers.md` in the corresponding section.

Task 7:

- Now we will modify the microarchitecture of the CPU to become two-way in-order superscalar by changing the `config/cs425_minor7.py` configuration of the CPU found in lines 80-112. Study carefully and adapt the widths and limits to allow for two instructions flowing through the pipeline per cycle.
- Run with `run7.sh` all three versions of the `simple_for` code with unpipelined functional units (`task7_1, task7_2, task7_3`) and with pipelined functional units (`task7_4, task7_5, task7_6`).
- View the pipeline behavior, report the performance numbers (cycles, instructions, CPI), and discuss your observations on the two-way in-order superscalar pipeline performance compared to the previous tasks in the text file called `answers.md` in the corresponding section.

Assignment Delivery

Commit and push the following files on your designated gitlab repo:

- `answers.md` with your report and comments
- The `code` folder with the multiple versions of the `simple_for.c` and `.dump`
- The `config` folder with the multiple versions of the `cs425_minor.py` configurations
- All the task (`task1, task2`, …) folders generated by the simulations
- Any other files or scripts you consider useful

Indicative git commands:

- git status (see the changes)
- git config user.email csdXYZW@csd.uoc.gr
- git config user.name "Name Surname"
- git add answers.md task* config* code* (add all related folders)
- git commit -m “sim1 delivery” (commit and message)
- git push (use your username and password)

Make sure that the push was successfully done and that your files have been uploaded on the gitlab repo.