# EIE2 Instruction Set Architecture & Compiler (IAC)

## Team Project - RISC-V RV32I Processor

***Peter Cheung, V1.1 - 29 Nov 2022***

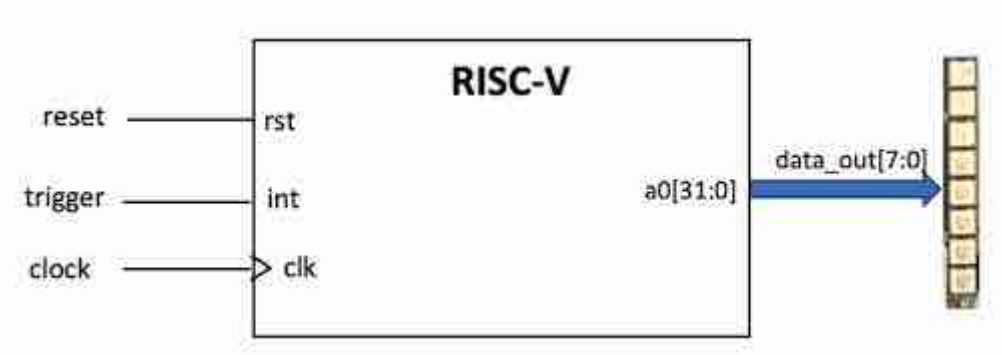
## Objectives

- To learn RISC-V 32-bit integer instruction set architecture
- To implememnt a single-cycle RV32I instruction set in a microarchitecture
- To implemententing the F1 starting light algorithm in RV32I assembly language
- To verify your RV32I design by executing the F1 light program
- As stretched goal, to implement a simple pipelined version of the microarchitecutre
- As a further stretched goal, add data cache to the pipelined RV32I, or
- As an alternative stretched goal, implement the RV32IM instruction set by adding multiply and divide hardware and instructions, or
- As another alternative stretch goal, add the simplified Wishbone bus to your RV32I or RV32IM processor

## Learning the RV32I Instruction Set

> Before you start the hardware design, every team member should learn the RV32I in some detail by jointly creating your team's assembly language program to implement the F1 starting light algorithm from Lab 3 in RV32I instructions.

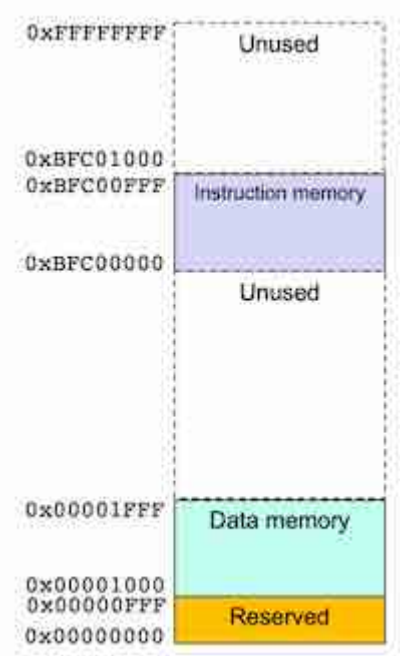
Your implementation **MUST** use at least one subroutine so that you must include the **Jump and Link (JAL)** instruction. Your team's F1 program, together with another REFERENCE program from me will be used to test your processor implementation.




Unlike Lab 3, the clock signal does not control a hardware counter or state machine directly. Instead it is used to clock the RISC-V processor to execute one instruction. The Reset signal also resets only the processor to start the program, and is not used to reset counters or a state machine. The trigger signal is used to tell RISC-V when to start the F1 light sequence. How it is implemented in the RISC-V is not defined.

You can decide, for example, that the trigger is automatic -- as soon as the program starts, the F1 light sequence is triggered. Alternatively you may implement a simple interrupt mechanism so that when trigger is asserted (say from low to high), RISC-V will start running the F1 light subroutine. The starting address of the interrupt handling routine is stored in an address location (called the interrupt vector).

You should also write your assembly language program using the following memory map.




This memory map is chosen to help debugging your design later. What is the size of the data and instruction memory specified by this map?

---

## Single-Cycle RV32I Design

---

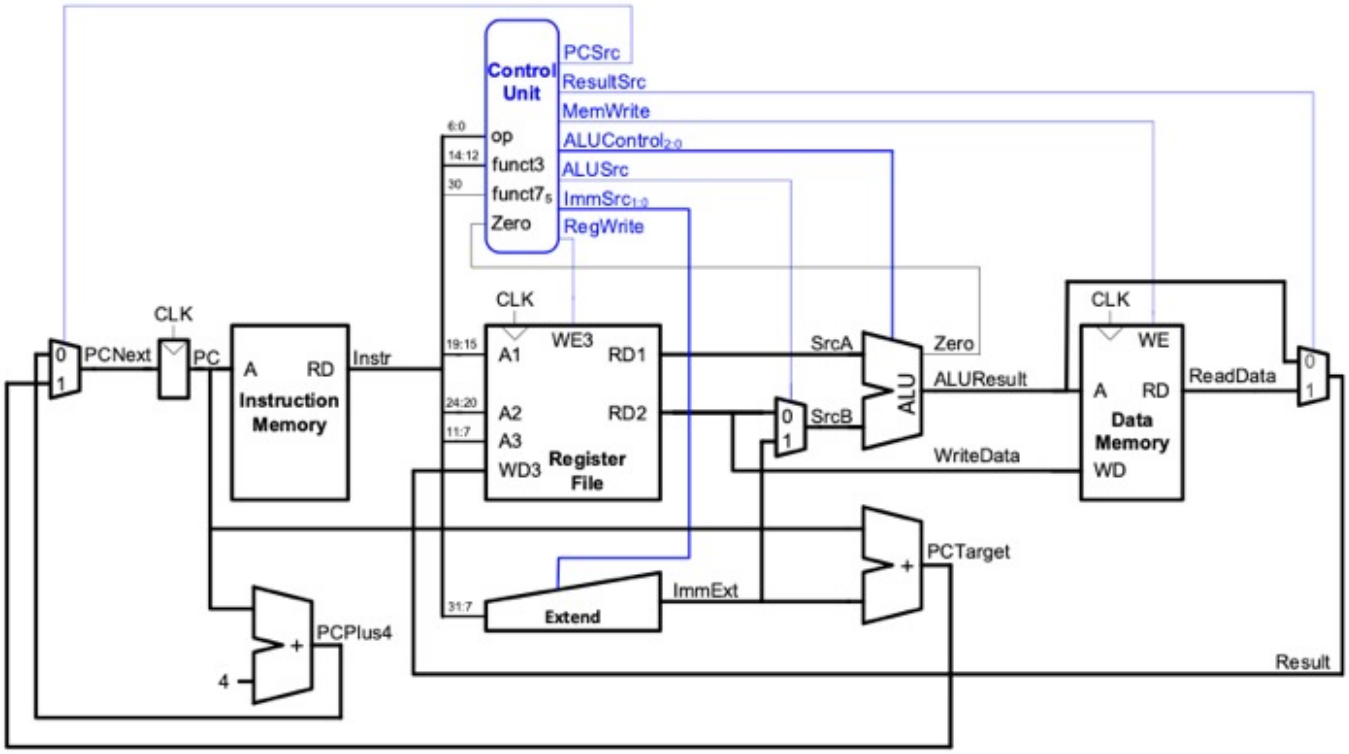
> This is the basic goal for every team - to implement the basic RV32I instruction set by extending your Reduced RISC-V design in Lab 4.

You do not need to implement every instruction in the RV32I instruction set, but you must implement the JAL instruction so that you an run code that uses subroutines.

To simplify matter, you should assume that you have separate instruction and data memory.

Similar to Lab 4, you must divide the task into roughly equal compnents, and each student will then be responsbile for one components. You can continue your role as in Lab 4 or, to diversify your learning by deliberately assigning a different component to your team members. The assessment of this project coursework will be mostly based on individual contributions with a smaller component based on the team's success. Details on assessment and deliverable are provided later in this project brief.

The testbench and test protocol for this processor is to provide evidence that your team's design is working for your team's F1 starting light program. You must also show evidence that your design also works for the REFERENCE program that I will provide later.




## Stretch Goal 1: Pipelined RV32I Design

> Once finished the basic goal, if your team have time, modify the the single-cycle processor to a pipelined processor.

You should handle data and control hazards in software - by identifying and inserting NOPs, or by re-ordering instructions to avoid hazards. You are not expected in implement hardware hazard detection, forwarding/bypassing or stalling hardware.

As before, make sure that your design is working by successfully running the team's version of the F1 light program.

## Additional Stretch Goal 1: Adding Data Memory Cache

As an additional stretch goal to the pipelined processor, you may also add data cache to your data memory. This is of course a "toy" exercise because your data memory is already as single-cycle memory and is very fast. Adding cache memory may make this slower, not faster. However, in real designs, data memory could be quite slow. Then, adding cache memory will help performance. Nevertheless, you may learn how cache memory works by implement it as an addition to your pipelined processor.

The data cache capacity should be no more than 256 bytes.

## BEYOND THIS COURSEWORK 1: Adding Multi-Cycle Integer Multiply/Divide Instructions

If you want to go even further than required by this courework, you may implement multi-cycle integer multiply and divide units to implement the RV32IM version of the RISC-V processor. This is beyond the taught materials in lectures and therefore you will have to learn the stuff yourself.

While it is possible to directly use `*` and `/` operators in SystemVerilog to multiply and divide, these can generate large amounts of hardware and must execute within one cycle as they are combinational. Often, this will introduce a new critical path in your CPU. Since the maximum clock speed of your CPU is determined by the section of the pipeline which runs the slowest, using such operators with 32 bit operands could significantly impact your maximum clock speed. As such, multiplication and division should be performed over several cycles.

## BEYOND THIS COURSEWORK 2: Adding Wishbone Bus interface

Another challenge that goes beyond the scope of his module is standard bus interface. If you want, you may choose to implement the Wishbone Bus interface between your CPU and your instruction and data memory.

In the real world, it's likely that your CPU will have to interoperate with components and IP cores that are built by other hardware designers. To do this, they need a common protocol to facilitate data exchange. There are many proprietary protocols for doing this such as Intel's Avalon and ARM's AXI, however they are significantly more complex than Wishbone B4. Wishbone also has the advantage that it is open-source.

Specifically you will need to implement the following parts of the Wishbone B4 specification:

- *3.2.1 Classic standard SINGLE WRITE Cycle* on page 40
- *3.2.3 Classic standard SINGLE READ Cycle* on page 43 and 44

You can ignore signals starting with `tga_`, `tgc_` and `tgd_` as these are not required for the operation of the protocol.

The Wishbone B4 specification can be found here

## Deliverable

All deliverables must be via your Team's project coursework repo via this link. The name of the repo has your team number at the end, and is private to your team, but accessible by myself and my TAs. All deliverables **must be** on the repo by ***mid-night Friday 16 December*** when all coursework team repos will be frozen.

Deliverables must include the following:

1. A `README.md` file in the root directory that briefly describe what your team has achieved. This is a **joint statement** for team and must include an agreed statement on the contributions by each member of the team.
2. Each individual's **personal statement** explaining what they contributed, reflection about what you have learned in this project, mistakes you made, special design decisons, and what you might do differently if you were to do it again or have more time. This statement must be succinct and to the point, yet must include sufficient details for me to check against the push history of the repo so that any claims can be verified. Including links to a selection of specific commits which demonstrate your work is advised. If you work with another member of your group on a module, make sure to give them co-author credit. Additionally, try to make meaningful commit messages.
3. A folder called `rtl` with the source of your processor. If you have multiple versions due to the stretched goals, you may use multiple branches. Your `README.md` file must provide sufficient explanation for the assessor to understand what you have done and how to find your work on all branches you wish to be assessed. The `rtl` folder should also include a `README.md` file listing who wrote which module/file.
4. A `test` folder with your F1 program. The folder must also contain test results for your processor successfully executing the F1 program and my reference program.

You must also provide a Makefile or a shell script that allows the assessor to build your processor model and run the testbench to repeat what you have done.

---

## Assessment Criteria

---

Assessment for this coursework, which accounts for 20% of the IAC module, is divided into two components:

1. Team achievement (40%) - This component of the marks is common to all team members and is dependent on the overall achievement of the team.
2. Individual achievement (60%) - This component of the marks is awarded to individual student based on declaration by the team of the individual contribution, with verification based on evidence (e.g. based on the git commit and push profile of an individual), individual account of his/her contributions and reflections, and the actual deliverables by the individual in terms of SystemVerilog, C++ codes and/or test results.

This table shows the level of team's achievement and the approximate range of grade to be awarded.

| Team Achievements | Grade Range |
|---|---|
| Verified pipelined RISC-V with data cache | A++ |
| Verified pipelined RISC-V only | A+ to A |
| Verified completion of single-cycle RISC-V | A- to B |
| Partially verified single-cycle RISC-V | B- to C |
| Attempted but unfinished single-cycle RISC-V | C- to D |

Here are the criteria on which individual assessment will be based.

| Individual's Evaluation |
| --- |
| Quality of design evidenced by SystemVerilog and/or C++ code and/or test results |
| Engagement and contribtuions evidenced by Team's statement of contributions and github profile data |
| Individual's mastery of the module evidenced by the individual's account and reflections |
| Peer's assessment as evidenced by the Final Team's survey |

## Tips

- Agree a coding style between your team and stick to it. This document contains many suggested practices which are good to follow.
- Suffix module inputs and outputs with `_i` or `_o` respectively. This will make it much easier to understand what you are looking at in Gtkwave.
- Setup a `.gitignore` file to prevent yourself from committing generated files, the `obj_dir` directory and `vcd` files into Git
- Review each file before you stage and commit it into Git. You will catch many errors this way. VS Code has in-built tooling for Git which you should try to get familiar with.
- Write useful commit messages
- Work on your own branches on Git and get other members of your team to review your code before merging it in to the main branch