EE/CE 321L/330L

Computer Architecture Lab

Lab Project

Version – Monday 2nd February 2026

Contents

Lab Exercises

1 Introduction to Computer Architecture Lab **4**
1.1 Purpose of the Lab . . . 4
1.2 Course Context and Learning Philosophy . . . 4
1.3 Overview of the Lab Sequence . . . 4
1.4 Why RISC-V? . . . 5
1.5 From Programs to Hardware . . . 5
1.6 Combinational and Sequential Circuits . . . 6
1.7 Clock and Reset Signals . . . 7
1.8 Synthesis-Centered Learning Using Vivado . . . 8
1.9 Timing Analysis, Clock Capture, and Slack . . . 8
1.10 XDC Constraints and Pin Assignments . . . 8
1.11 Synthesis Warnings and Debugging (Expanded) . . . 9
1.12 From Synthesis to Implementation . . . 10
1.13 Synthesis Cookbook: Recommended Workflow . . . 10
1.14 Vivado Synthesis & Timing Terminology . . . 10
1.15 Professional Practice and Assessment . . . 12

2 Practical Information **13**
2.1 Course Learning Outcomes . . . 13
2.2 Lab Schedule and Assessments . . . 13
2.3 Groups . . . 14
2.4 Approval of Exercises . . . 14
2.5 Project Evaluation . . . 15
2.6 Lab Rubrics . . . 16

3 Version Control: Git and GitHub **19**
3.1 What is Version Control? . . . 19
3.2 Why is Version Control Needed? . . . 19
3.3 Available Version Control Tools . . . 19
3.4 What is Git and Why Use It? . . . 20
3.5 What is GitHub? . . . 20
3.6 How are Git and GitHub Different? . . . 20
3.7 Focusing on Git: workflow and commands . . . 21
3.7.1 What is a Repository? . . . 21

3.7.2 Getting Started with Git 21
3.7.3 Initializing a Repository 22
3.7.4 Cloning an Existing Repository 22
3.7.5 Add Files to the Staging Area 24
3.7.6 Commit Changes 25
3.7.7 Link to Remote Repository 26
3.7.8 Push Local Code to GitHub 28
3.7.9 Fetch Changes from Remote 29
3.7.10 Pull Changes from Remote 30
3.7.11 Checking Differences or Changes made in the Files 31
3.7.12 Create and Switch Branches 33
3.7.13 Merging Branches 34
3.7.14 Optional: Remove a Tracked File 37
3.7.15 Summary of Essential Git Commands 38

Lab 1: Getting Started with RISC-V (Assembly Language) in VS Code 40

Lab 2: Implementing Decision Instructions in RISC V Assembly Language 56

Lab 3: Implementing Jump and Return Instructions in RISC V Assembly Language 62

Lab 4: Implementing Nested Procedures and Sorting in RISC V Assembly Language 69

Lab 5: Pre lab - Synthesizable vs Non-Synthesizable Verilog 75

Lab 5: Designing FSM Using FPGA Switches and LEDs 79

Lab 6: Design and FPGA Implementation of the ALU for a Single-Cycle RISC-V Processor 87

Lab 7: Design and FPGA Implementation of the Register File for a Single-Cycle RISC-V Processor 91

Lab 8: Design and FPGA Implementation of Memory System with Address Decoding for a Single-Cycle RISC-V Processor 96

Lab 9: Design and FPGA Implementation of the Control Path for a Single-Cycle RISC-V Processor 102

Lab 10: State-Based Control Flow Using RISC-V Assembly 106

Open Ended Lab : Design and FPGA Implementation of a Single-Cycle RISC-V Processor 111

Project: Assembly-Level Programs Execution on a Hardware-Implemented 32 bit RISC-V Processor 115

Introduction to Computer Architecture Lab

1.1 Purpose of the Lab

The Computer Architecture Lab complements the lecture course by providing hands-on experience with both software-level and hardware-level aspects of processor design. Rather than treating a processor as an abstract black box, students will directly observe how instructions are executed by hardware through design, simulation, synthesis, and FPGA implementation.

The primary objective of this lab is to guide students from RISC-V assembly programming to the design and implementation of a complete single-cycle RISC-V processor on an FPGA, reinforcing the connection between programs and the hardware that executes them.

1.2 Course Context and Learning Philosophy

Computer architecture spans multiple abstraction levels, including:

- Assembly language programming
- Digital logic and finite state machines
- Register-transfer-level (RTL) hardware design
- Timing and physical implementation constraints

This lab follows a bottom-up learning philosophy:

1. Understand what instructions do (assembly)
2. Understand how hardware implements them (datapath and control)
3. Understand why designs succeed or fail on real hardware (timing and synthesis)

Students are expected to reason not only about functional correctness, but also about clocking, timing, synthesis warnings, and FPGA constraints.

1.3 Overview of the Lab Sequence

The lab consists of 11 labs and a final integration project, divided into two phases.

Phase I: RISC-V Assembly Language (Labs 1–4)

The first four labs focus on RISC-V assembly programming, covering:

- Arithmetic and logical instructions
- Memory access and addressing
- Branches, loops, and conditionals
- Procedures, recursion, and stack usage

These labs define the exact behavior that the processor hardware must support.

Phase II: Hardware Design and FPGA Synthesis (Labs 5–11)

Beginning in Lab 5, the focus shifts to hardware design using Verilog HDL and FPGA synthesis using Xilinx Vivado.

Students incrementally design:

- Finite state machines
- An arithmetic logic unit (ALU)
- A register file
- Memory interfaces
- Control logic
- A complete single-cycle RISC-V processor

Each lab adds a new hardware component and integrates it into a growing processor design.

1.4 Why RISC-V?

RISC-V is an open, modern instruction set architecture well suited for education because it:

- Has simple and regular instruction formats
- Uses a small base ISA (RV32I)
- Clearly separates datapath and control
- Is widely used in academia and industry

The same RISC-V assembly programs written in early labs are executed by the processor built in later labs.

1.5 From Programs to Hardware

Each RISC-V instruction activates specific hardware structures:

- `add`: ALU and register file

- `lw`: ALU, data memory, and write-back logic
- `beq`: ALU comparison and PC control
- Procedure calls: stack pointer, memory, and PC redirection

Understanding this mapping is the central goal of computer architecture.

1.6 Combinational and Sequential Circuits

All digital hardware in this lab is built from combinational and sequential circuits.

Combinational Circuits

Combinational circuits produce outputs that depend only on current inputs and do not store state.

Examples include:

- ALUs
- Multiplexers
- Decoders

Example: Combinational ALU Logic (Verilog)

```
always @(*) begin
    case (alu_ctrl)
        3'b000: result = a + b;
        3'b001: result = a - b;
        3'b010: result = a & b;
        3'b011: result = a | b;
        default: result = 32'b0;
    endcase
end
```

Key properties:

- Uses `always @(*)`
- All outputs assigned in every path
- No clock or memory elements

Incomplete assignments result in inferred latches, which are usually design errors.

Sequential Circuits

Sequential circuits store state and update only on clock edges.

Examples include:

- Registers
- Program counter
- FSM state registers

Example: Register with Synchronous Reset (Verilog)

```
always @(posedge clk) begin
    if (reset)
        q <= 32'b0;
    else
        q <= d;
end
```

Key properties:

- Triggered by `posedge clk`
- Uses non-blocking assignments (<=)
- Reset ensures a known initial state

All processor state in this lab is implemented using sequential logic.

1.7 Clock and Reset Signals

Clock Signal

The clock determines when state changes occur. All designs in this lab:

- Use a single global clock
- Are fully synchronous
- Avoid gated or derived clocks

The clock period limits the maximum operating frequency.

Reset Signal

The reset signal initializes all state-holding elements. Resets:

- Establish predictable startup behavior
- Initialize FSMs and registers
- Prevent undefined states

This lab uses synchronous resets consistently across all modules.

1.8 Synthesis-Centered Learning Using Vivado

Vivado synthesis translates RTL into FPGA hardware:

- Look-Up Tables (LUTs)
- Flip-flops
- Block RAMs
- Routing resources

Students are required to inspect RTL schematics, technology schematics, and resource utilization reports.

1.9 Timing Analysis, Clock Capture, and Slack

Vivado performs static timing analysis (STA) to verify that data launched from one register arrives in time for capture by the next and that setup and hold constraints are met.

Key concepts include launch register, capture register, critical path, and setup and hold time.

Slack measures timing margin:

- Positive slack: timing met
- Negative slack: timing violation

1.10 XDC Constraints and Pin Assignments

FPGA designs require XDC (Xilinx Design Constraints) files to define clocks and I/O behavior.

Example XDC File

```
## Clock definition (100 MHz)
create_clock -period 10.0 -name sys_clk [get_ports clk]

## Reset input
set_property PACKAGE_PIN V17 [get_ports reset]
set_property IOSTANDARD LVCMOS33 [get_ports reset]

## LED output
set_property PACKAGE_PIN U16 [get_ports led]
set_property IOSTANDARD LVCMOS33 [get_ports led]
```

Without correct constraints, timing analysis and hardware behavior are unreliable.

1.11 Synthesis Warnings and Debugging (Expanded)

During synthesis and implementation, Vivado generates warnings that must never be ignored. These warnings often indicate design flaws that may not appear in simulation but can cause incorrect hardware behavior.

Inferred Latches

Latches occur when combinational outputs are not assigned in every execution path. They introduce unintended memory and complicate timing analysis.

Multiple Drivers

Multiple drivers occur when a signal is assigned in more than one place, resulting in undefined hardware behavior.

Unused or Trimmed Logic

Trimmed logic indicates disconnected or unused signals and often points to integration errors.

Unconstrained Clocks

Unconstrained clocks prevent meaningful timing analysis and may cause hardware failure.

Combinational Loops

Combinational loops cause unstable logic and must be broken using registers.

Width Mismatch and Truncation

Width mismatches cause loss of data and subtle functional bugs.

Reset-Related Warnings

Improper reset usage leads to undefined startup behavior.

Timing Violations and Negative Slack

Negative slack indicates that the design cannot operate reliably at the target frequency.

1.12 From Synthesis to Implementation

After synthesis, Vivado performs placement, routing, and final timing verification before generating the FPGA bitstream.

1.13 Synthesis Cookbook: Recommended Workflow

1. Write clean RTL
2. Separate combinational and sequential logic
3. Use consistent clock and reset strategy
4. Simulate thoroughly
5. Apply XDC constraints
6. Inspect synthesis and timing reports
7. Fix warnings and timing violations
8. Implement and test on FPGA hardware

1.14 Vivado Synthesis & Timing Terminology

Term	**Explanation**
RTL (Register Transfer Level)	A hardware abstraction that models digital systems using registers, combinational logic, and clocked data transfers. RTL describes how data moves between registers on clock edges and how logic operates between them. Vivado synthesizes RTL written in Verilog, VHDL, or SystemVerilog.
HDL (Hardware Description Language)	A specialized programming language used to model digital hardware behavior and structure. Vivado supports Verilog, VHDL, and SystemVerilog for describing RTL designs that can be simulated and synthesized.
Synthesizable RTL	The subset of HDL constructs that can be converted into physical hardware. Constructs such as flip-flops, combinational logic, and finite state machines are synthesizable, while simulation-only features like delays (#10) are ignored by synthesis.
Vivado Synthesis	The process in Vivado that converts RTL code into a technology-specific netlist. During synthesis, logic is optimized for performance, area, or power based on constraints and FPGA architecture.
Technology Mapping	The stage where synthesized logic is mapped onto FPGA primitives such as LUTs, flip-flops, DSP blocks, and BRAMs supported by the target Xilinx device.
Netlist	A structural description of the design consisting of logic elements and their interconnections. The netlist is the output of synthesis and serves as the input to implementation.

Term	Explanation
Gate-Level Netlist	A low-level netlist composed only of logic gates, registers, and interconnects. It is used for post-synthesis simulation and detailed timing analysis.
Clock Constraint (create_clock)	A timing constraint defined in the XDC file that specifies the clock period or frequency. It guides Vivado timing analysis and optimization. Example: `create_clock -period 10 [get_ports clk]`
XDC File	Xilinx Design Constraints file used to specify clocks, I/O pin locations, timing exceptions, and physical constraints required for correct synthesis and implementation.
Static Timing Analysis (STA)	A mathematical timing verification technique that checks whether all signal paths meet setup and hold timing requirements without requiring simulation.
Timing Path	A signal path analyzed by STA, typically from a launching register to a capturing register, or between an input and output port.
Setup Time	The minimum time data must be stable before a clock edge so that it is reliably captured by a flip-flop. Violations cause incorrect data sampling.
Hold Time	The minimum time data must remain stable after a clock edge. Hold violations are often caused by short combinational paths and clock skew.
Slack	The margin between required arrival time and actual signal arrival time. Positive slack indicates timing is met; negative slack indicates a violation.
Worst Negative Slack (WNS)	The most severe setup timing violation in the design. WNS determines whether the design can operate at the target clock frequency.
Total Negative Slack (TNS)	The sum of all negative slack values across failing paths. TNS indicates how widespread timing violations are in the design.
Critical Path	The longest-delay timing path in the design. This path limits the maximum achievable clock frequency (Fmax).
Fmax	The maximum clock frequency at which the design can operate without timing violations. It is inversely proportional to the critical path delay.
Fan-Out	The number of logic elements driven by a single signal. High fan-out increases routing delay and may degrade timing performance.
Optimization	The process by which Vivado improves design performance, area, or power consumption through logic restructuring, retiming, and resource sharing.
Resource Utilization	A report summarizing the number of LUTs, flip-flops, BRAMs, DSPs, and other FPGA resources consumed by the design.
Place and Route (Implementation)	The Vivado stage that assigns logic elements to physical locations on the FPGA and routes signals between them considering timing and congestion.
Post-Synthesis Simulation	Simulation performed using the synthesized netlist to verify functional correctness before physical placement and routing.
Post-Implementation Timing Analysis	Final timing verification after placement and routing, accounting for real routing delays and clock skew.
Multicycle Path	A timing exception allowing a signal path to take multiple clock cycles to complete, reducing unnecessary timing constraints.

Term	**Explanation**
False Path	A timing path that never occurs during real operation and is excluded from timing analysis to prevent false violations.
Clock Skew	The difference in clock arrival times at different registers caused by routing delays. Excessive skew can cause setup or hold violations.
Metastability	A condition where a flip-flop enters an unstable state due to asynchronous input changes near the clock edge, potentially causing unpredictable behavior.
Synthesis Report	A Vivado-generated report summarizing optimization results, estimated timing, and resource utilization after synthesis.
Implementation Report	The final report produced after place and route, providing detailed timing, power, and utilization information under real physical conditions.

1.15 Professional Practice and Assessment

Assessment emphasizes functional correctness, verification quality, synthesis and timing awareness, and clear documentation. Students are expected to treat this lab as a real engineering design experience.

Practical Information

2.1 Course Learning Outcomes

CLO	Description	LDL
CLO-1	To design and develop digital design modules of a RISC-V processor using Verilog HDL and simulate them using simulators (ModelSim® / EDA Playground).	COG-5
CLO-2	To utilize the RISC-V assembly instruction set for implementation of case structures, loops, and functions through an online simulator.	COG-3
CLO-3	Design and Synthesis of a Single Cycle RISCV Processor on Basys3 board.	COG-5

2.2 Lab Schedule and Assessments

The lab has 11 lab experiments and a final project. Weightages of each assessment are given below.

Week#	Lab Name	CLO	Weightage (%)
1	Lab 1: Getting Started with RISC-V (Assembly Language) in VS Code	CLO-2	3
2	Lab 2: Implementing Decision Instructions in RISC V Assembly Language	CLO-2	5
3	Lab 3: Implementing Jump and Return Instructions in RISC V Assembly Language	CLO-2	5
4	Lab 4: Implementing Nested Procedures and Sorting	CLO-2	5
5	Lab 5: Designing FSM Using FPGA Switches and LEDs	CLO-1	7
6	Lab 6: Design and FPGA Implementation of the ALU for a Single-Cycle RISC-V Processor	CLO-1	7
7	Lab 7: Design and FPGA Implementation of the Register File for a Single-Cycle RISC-V Processor	CLO-1	7
8	Lab 8: Design and FPGA Implementation of Memory System with Address Decoding for a Single-Cycle RISC-V Processor	CLO-1	7
9	Lab 9: Design and FPGA Implementation of the Control Path for a Single-Cycle RISC-V Processor	CLO-1	7
10	Lab 10: FSM Implementation Using Assembly Language	CLO-1	7
11	Open Ended Lab: Design of a Single-Cycle RISC-V Processor	CLO-3	10
12-14	Unsupervised Lab Sessions	-	-
15	Project: Single Cycle Processor	CLO-3	30

2.3 Groups

Each lab will be conducted in groups of two students and the project will be completed in groups of three. The project group team registration will happen during the third lab session.

2.4 Approval of Exercises

After finishing each exercise, students must show their results to the research assistant to get approval for grading. If a group cannot get approval on the lab day, a late penalty will apply for getting it later. If the group does not get approval within 7 days (before the next lab), they will receive a zero for that lab.

In addition, Habib University's exam rules and policies on submission and plagiarism will be followed.

2.5 Project Evaluation

At the end of the semester, all lab and project code must be submitted by pushing it to the assigned Git repositories on the LMS. Students are expected to use Git properly, including writing clear commit messages and making regular updates throughout the course.

Groups are required to present their project during a scheduled demonstration and oral presentation with the course instructor/RA. Each group will have 15 minutes for their presentation. The source code must be fully uploaded before the presentation. Additionally, a written project report must be submitted before the evaluation day. The report should clearly describe the project objectives, design, implementation and results of all required tasks.

Although formal presentation slides are not required, groups are encouraged to prepare a short (3–5 minute) oral presentation to explain their work and help the evaluators understand it. This will be followed by a question and answer session.

Approval of exercises shows that key parts were working at some point but does not guarantee full marks; this will be considered if problems arise during evaluation.

The contribution of each group member will be individually assessed during the project evaluation. All members are expected to actively participate in the evaluation and demonstrate understanding of all assigned tasks. Insufficient contribution or inability to answer questions may result in a reduced viva score, which will be used as a scaling factor for the final project grade.

Additional details about the evaluation process will be provided closer to the evaluation date.

Computers and Equipment

The lab is equipped with computers that have all the required software pre-installed. These computers will mainly be used for development, reading manuals, and interfacing with FPGA. Installing additional software is not allowed. If you think any essential software is missing and would benefit all students, please contact the research assistant to request its installation.

Specialized or group-specific tools and software must be run on your personal laptop. Do not save project files or code on lab computers, as these may be upgraded during the semester without notice, causing loss of all local data. Instead, use the git and github version control system to securely store your work remotely.

2.6 Lab Rubrics

ID	Assessment Element	Level 1: Unsatisfactory	Level 2: Developing	Level 3: Good	Level 4: Exemplary
LR1	Circuit Layout	Connections between circuit components are mostly wrong. Circuit layout is cluttered. Needs guidance to make correct equipment/component connections.	Few of the connections made between circuit components are wrong. Circuit layout is not neat and clean as per standards.	Circuit layout and connections are correct but not all connections are neat and clean as per standards.	Neat, clean and correct connections of all circuit components/equipment are made as per standard circuit diagram.
LR2	Program/Code/Simulation Model/Network Model	Program/code/simulation model/network model does not implement the required functionality and has several errors. Student is not able to utilize even the basic tools of the software.	Program/code/simulation model/network model has some errors and does not produce completely accurate results. Student has limited command on the basic tools of the software.	Program/code/simulation model/network model gives correct output but not efficiently implemented or implemented by computationally complex routine.	Program/code/simulation/network model is efficiently implemented and gives correct output. Student has full command on the basic tools of the software.
LR3	Troubleshooting	Unable to identify the fault/minimal effort shown in troubleshooting.	Able to identify the fault but unable to remove it.	Able to identify the fault but partially removes it.	Able to identify the fault and takes necessary steps and actions to correct it.
LR4	Data Collection	Measurements are incomplete, inaccurate and imprecise. Observations are incomplete or not included. Symbols, units and significant figures are not included.	Measurements are somewhat inaccurate and imprecise. Observations are incomplete or vague. Major errors are there in using symbols, units and significant digits.	Measurements are mostly accurate. Observations are generally complete. Minor errors are present in using symbols, units and significant digits.	Measurements are both accurate and precise. Data collection is systematic. Observations are very thorough and include appropriate symbols, units and significant digits and task completed in due time.
LR5	Results & Plots	Figures/graphs/tables are not developed or are poorly constructed with erroneous results. Titles, captions, units are not mentioned. Data is presented in an obscure manner.	Figures, graphs and tables are drawn but contain errors. Titles, captions, units are not accurate. Data presentation is not too clear.	All figures, graphs, tables are correctly drawn but contain minor errors or some details are missing.	Figures/graphs/tables are correctly drawn and appropriate titles/captions and proper units are mentioned. Data presentation is systematic.
LR6	Calculations	Formulae used and/or calculations are incorrect. Units are not mentioned with results.	Formulae used are correct but there are few mistakes in calculation which lead to incorrect results.	Formulae used and end results are correct. Complete working steps are not shown. Proper units are not mentioned with results.	Formulae used, end results and all calculations are correct with all intermediate steps clearly shown. Units are mentioned with results.
LR7	Viva	Response shows a complete lack of understanding of the assigned/completed task.	Response shows shallow understanding of the assigned task.	Response shows substantial understanding of the assigned task.	Response shows complete understanding of the completed task. Student is able to explain all related concepts.
LR8	Equipment/Instrument Handling	Inappropriate handling of the tools and equipment with minimal accuracy.	Appropriate handling of some of the tools and equipment.	Appropriate handling of most of the tools and equipment.	Appropriate handling of all the tools and equipment.
LR9	Report	All the in-lab tasks are not included in report and/or the report is submitted too late.	Most tasks are included in report but not well explained. Necessary figures/plots are not included. Report is submitted after due date.	Good summary of most in-lab tasks included. Work supported by figures/plots with explanations. Report submitted timely.	Detailed summary of in-lab tasks provided. All tasks included and explained well. Data clearly presented including necessary figures, plots and tables.
LR10	Analysis	Results are not interpreted or analyzed incorrectly.	Analysis of obtained results is incomplete. Conclusions are not drawn.	Results are analyzed and concluded but not well explained; justification is not provided with reasoning.	Results are well analyzed supported with reasons. Good comparison between theoretical and experimental results with correct and useful conclusion.
LR11	Design	Proposed design is unsubstantiated or does not satisfy most constraints. Breakdown into features is incomplete and still at lower resolution.	Justification is weak; satisfies some constraints. Breakdown missing some features and resolution inappropriate at some places.	Proposed design is substantiated (preferably through analysis) and satisfies most constraints. Breakdown seems complete, but resolution inappropriate at some places.	Proposed design is substantiated (preferably through analysis) and satisfies all constraints. Breakdown complete and at appropriate resolution given students' level.

Affective Domain Rubric

ID	Assessment Element	Level 1: Unsatisfactory	Level 2: Developing	Level 3: Good	Level 4: Exemplary
AR7	Report Content/Code comments	Most of the questions are not answered / figures are not labelled/ titles are not mentioned / units are not mentioned. No comments are present in the code.	Some of the questions are answered, figures are labelled, titles are mentioned and units are mentioned. Few comments are stated in the code.	Majority of the questions are answered, figures are labelled, titles are mentioned and units are mentioned. Comments are stated in the code.	All the questions are answered, figures are labelled, titles are mentioned and units are properly mentioned. Proper comments are stated in the code.

Open Ended Lab Rubric

OE	Assessment Elements	Level 1: Unsatisfactory	Level 2: Developing	Level 3: Good	Level 4: Exemplary
OE1	Problem Definition	Fails to clearly define the problem or research question.	The problem has been defined to some extent but lacks precision or relevance.	The problem or research question has been defined clearly and contextually.	The problem has been defined clearly. Research has been done with precision and relevance.
OE2	Experimental Design	The proposed design is unsatisfactory and does not satisfy most of the given constraints. The breakdown of system into features is incomplete and still at lower resolution.	The justification for the proposed design is weak, and it only satisfies some constraints. The system is missing some features or resolution is not apparent in some places.	The proposed design is developed well through proper analysis, research and statistics that show the breakdown of system into components is complete, but resolution is not apparent in some places.	The proposed design is developed thoroughly through proper analysis, research and statistics that show the breakdown of system into appropriate resolution. The features seem complete and appropriate to satisfy all given constraints.
OE3	Data Collection	Measurements are incomplete, inaccurate and imprecise. Observations are incomplete or not indicated. Symbols, units and significant figures are not used.	Measurements are somewhat inaccurate or imprecise. Observations are missing or sparse. Major errors are there in the use of symbols, units and significant figures.	Measurements are mostly accurate. Observations are generally complete. Minor errors are present in using symbols, units and significant figures.	Measurements are both accurate and precise. Data collection is systematic. Observations are complete. No significant errors in symbols, units and significant figures.

OE4	Analysis & Interpretation	Fails to analyze data effectively or identify meaningful trends or interpretations.	The analysis of data is missing steps or important trends have been missed or provided limited insight.	The data has been analyzed effectively, identifying trends and drawing reasonable conclusions.	The data has been analyzed thoroughly, showing insight and drawing clear, logical, and well-supported conclusions.
OE5	Results, Plots and Report	Figures/graphs/tables are either not included or poorly constructed. Raw data presentation is unclear. Titles, captions, units are not accurate. Data presentation is either not included in the report or not present at all.	Figures, graphs and tables are drawn but contain errors. Titles, captions, units are not accurate. Data presentation is there, but most of the values are either not explained or not well explained.	All figures, graphs, tables are accurate. Titles and captions contain minor errors and are shown. Most of the required data is also presented and meets the prescribed requirements.	All figures/graphs/tables are accurate. Titles and captions are complete and appropriate. Data presentation meets all requirements and is explained in original and meaningful way to engage readers.
OE6	Reflection	Lacks meaningful self-evaluation or reflection on the experiment.	Offers limited self-evaluation without significant insights.	Reflects on the experiment, identifying areas of success and areas for improvement.	Demonstrates critical self-evaluation, identifying successes, weaknesses, and potential areas for refining the experimental approach.

Version Control: Git and GitHub

3.1 What is Version Control?

Version control is a system that helps track changes to files over time. It acts as a *save history* for your projects, allowing you to revert to previous versions, compare changes, and collaborate with others safely. Using a version control system is essential for efficiently managing embedded projects. It allows you to track changes, collaborate with others, and revert to earlier versions when needed. In this course, we recommend using **Git**, a widely adopted version control system, along with **GitHub** for online storage and collaboration.

3.2 Why is Version Control Needed?

In software development and other fields, files are often modified by multiple users or over time. Without version control:

- **Track progress:** Keep a detailed history of project changes.
- **Undo mistakes:** Restore previous versions when needed.
- **Collaboration:** Multiple contributors can work without conflicts.
- **Testing:** Makes it easy to experiment in isolated branches without affecting the main codebase.
- **Backup:** Ensures your work is safely stored and recoverable.

Version control solves these problems by managing file versions efficiently and providing collaboration tools.

3.3 Available Version Control Tools

Some popular version control systems include:

- CVS (Concurrent Versions System)
- Subversion (SVN)
- Mercurial
- **Git** (Distributed)

Among these, Git has become the most widely used due to its speed, flexibility, and distributed architecture.

3.4 What is Git and Why Use It?

Git is a distributed version control system designed for speed and efficiency, used both in industry and academia. Unlike centralized systems, Git allows every user to have a full copy of the repository history on their own computer, enabling faster operations and offline work. For this lab, we will focus on and use Git extensively. It allows users to:

- Track project changes locally
- Collaborate remotely using platforms like GitHub
- Use branching for safe experimentation
- Efficient handling of large projects.
- Branching and merging capabilities facilitate feature development.
- Integration with platforms like GitHub for remote collaboration.

3.5 What is GitHub?

GitHub is a web-based platform built on top of Git that provides remote hosting of Git repositories along with tools for collaboration, issue tracking, code review, and more. GitHub is a free cloud/remote storage option for Git, however it is not the only option. Paid third part servers such as Amazon and MS Azure also offer remote storage for Git. Key features of GitHub are:

- Online collaboration and code sharing
- Pull requests and code review
- Issue tracking and project management

3.6 How are Git and GitHub Different?

Feature	**Git**	**GitHub**
Type	Command-line tool	Web-based platform
Purpose	Version control	Hosting and collaboration
Internet Required	No	Yes
Example	`git commit`, `git merge`	Creating pull requests, viewing repositories

3.7 Focusing on Git: workflow and commands

For this lab, we will be using Git via command line. There are also graphical tools available but we will not be discussing them and use of such tools for getting lab approval is not allowed. Since we will be developing our firmware using VS Code, VS Code's add-on named "gitgraph" is the visualizing tool allowed in this lab.

3.7.1 What is a Repository?

A repository (repo) is a directory (commonly called folder in Windows) that contains your project files and version history. You can think of a repository like a project folder that not only contains your files but also has a time machine built in. You can look back at previous versions, see who made which changes, and collaborate with others without overwriting each other's work. There are two types of repositories.

- **Local repository:** Stored on your computer.
- **Remote repository:** Hosted on platforms like GitHub.

When you create a Git repository, Git internally creates a hidden folder named `.git`, which contains all the metadata and history of your project:

- Your commits (history)
- References (branches, tags)
- Staging area (index)
- Configuration settings

This structure allows you to:

- Create branches for features
- Merge different development lines
- Track who made changes and when
- Revert back to previous versions safely

3.7.2 Getting Started with Git

To begin using Git on your local machine, you need two things:

1. Install Git: `https://git-scm.com/downloads`
2. Set up your Git identity. Since everything is timestamped and tracks the user who made changes, setting up identity account is a mandatory requirement:

```
    git config --global user.name "Your␣Name"
    git config --global user.email "your.email@example.com"

```

Note that your local and GitHub account details can differ. However, whenever you will be pushing changes from local to GitHub, you will need to have a GitHub account, and inside the command terminal, you will be asked to provide your GitHub credentials. You can also set your remote account credentials globally on your laptop/PC. See Git documentation for this.

3.7.3 Initializing a Repository

To start tracking a project with Git, navigate to your project folder (e.g. mci_stm32f3_projects folder) and run:

```
cd your_project_directory
git init
```

This turns the current directory into a Git repository by creating the hidden `.git` folder.

3.7.4 Cloning an Existing Repository

If you have created a repository in lab and want to work on your laptop at home, you would like to copy the same project instead of re-creating it. To achieve this, you need GitHub, i.e. remote cloud storage. In the lab, you will push your local repository to the cloud. We will see this later on but assuming that your repository is already pushed to the cloud, use:

```
cd your_project_directory
git clone https://github.com/username/repo-name.git
```

This will:

- Download all files
- Set up the complete version history
- Link the local copy to the remote GitHub repository

Standard Way of Initializing a Repository

After you initialize a Git repository using `git init`, it is good practice to immediately set up two important files:

(a) `.gitignore` — to exclude unwanted files from version control

(b) `README.md` — to describe your project and how to use it

These files help make your repository clean, professional, and easier to understand for collaborators or future contributors (including yourself).

1. Create a `.gitignore` File

The `.gitignore` file tells Git which files or directories to ignore. These are usually files that:

- Are generated automatically (e.g., compiled binaries, log files)
- Contain machine-specific or personal configurations
- Should not be shared (e.g., API keys, secrets)

Example: Minimal `.gitignore` for a C Project

```
# Ignore compiled objects and binaries
*.o
*.out
*.exe

# Ignore build folder
/build/

# Ignore logs and editor backup files
*.log
*.swp
```

Commands to use:

```
touch .gitignore
# or manually create the file in your editor
```

After creating it, stage and commit it:

```
git add .gitignore
git commit -m "Add␣.gitignore␣to␣exclude␣compiled␣files"
```

2. Create a `README.md` File

The `README.md` file is a Markdown document that introduces and explains your project. It is the first thing users and collaborators will see on platforms like GitHub.

A Good `README.md` Usually Includes:

- Project name and short description
- How to install or compile the code
- How to use it (commands or examples)
- Contact or authorship info

Example: README.md for a C Calculator Project

```
# Simple C Calculator

A command-line calculator that supports basic and scientific operations.

## Compile

‘‘‘bash
gcc -o calc main.c math.c
```

3.7.5 Add Files to the Staging Area

Once the repository is created, you will create or modify files in your project. Git does not track them automatically. You must tell Git which files to track and include in the next commit. This is done using the `git add` command. The space where files are prepared before committing is called the **staging area** or **index**.

Syntax

```
git add <filename>
git add .
```

- `git add <filename>` adds a specific file.
- `git add .` stages all modified or new files in the current directory.

Example: C Programming Project

Suppose you are working on a C project with the following files:

- `main.c`
- `math.c`
- `math.h`
- `README.md`

After writing your code, you want Git to start tracking these files. Here's how:

```
git add main.c
git add math.c
git add math.h
git add README.md
```

Alternatively, to stage everything at once:

```
git add .
```

Tip: Check Staged Files

You can check what's currently in the staging area using:

```
git status
```

This command will show:

- Files staged for the next commit
- Untracked files (not yet added)
- Modified files not yet staged

Please note that all these command (in-fact all commands starting with `git` except `git init`) must be executed inside the repository folder. Once you have added the files to the staging area, you're ready to commit them using `git commit`.

3.7.6 Commit Changes

Once you've added files to the staging area using `git add`, the next step is to **commit** those changes. A commit in Git is like taking a snapshot of your project at a certain point in time. It saves the current state of the staged files into your **local** repository history.

Each commit should have a message that clearly describes what was changed and why.

Syntax

```
git commit -m "Your␣descriptive␣commit␣message"
```

Example: C Programming Project

Suppose you're working on a simple calculator written in C. After editing or creating the following files:

- `main.c`
- `math.c`
- `math.h`

As described earlier, you first stage the files:

```
git add main.c math.c math.h
```

Then commit them:

```
git commit -m "Add␣basic␣calculator␣functions␣and␣main␣program␣structure"
```

This commit will now be saved in the project's history. Git will track the fact that you added new files and what their contents were.

Why Commit Messages Matter

Commit messages are essential for:

- Understanding what changes were made
- Helping teammates follow your work
- Navigating through project history later

Good message examples:

- `"Fix bug in division by zero case"`
- `"Add unit tests for math functions"`
- `"Refactor main.c for better readability"`

Avoid vague messages like:

- `"Update"`
- `"Changes"`

View Commit History

To view a list of past commits:

```
git log
```

This shows:

- Commit IDs
- Author
- Date
- Commit messages

You can scroll through the history to understand how the project has evolved over time.

3.7.7 Link to Remote Repository

So far, all your Git operations have been local — everything is stored only on your computer. To collaborate with others or back up your work online, you need to connect your local repository to a **remote repository**, typically hosted on a platform like GitHub.

Step 1: Create a Remote Repository on GitHub

(a) Go to `https://github.com`

(b) Log in and click the + icon in the top-right corner

(c) Choose `New repository`

(d) Enter a repository name (e.g., `c-calculator`)

(e) **Do not initialize** with a README or license (since you already have local files)

(f) Click `Create repository`

Step 2: Connect Local to Remote

In your terminal, run the following command to add a remote named `origin`:

```
git remote add origin https://github.com/your-username/c-calculator.git
```

- Replace `your-username` with your GitHub username.
- Replace `c-calculator.git` with your repository name.

What does this do? This command tells Git, "Hey, I want to link my local repo to this online repo." The name `origin` is just a convention for the default remote.

Check the Connection

You can verify that your remote is set using:

```
git remote -v
```

This should show something like:

```
origin  https://github.com/your-username/c-calculator.git (fetch)
origin  https://github.com/your-username/c-calculator.git (push)
```

Fixing Mistakes (Optional)

If you accidentally entered the wrong URL or want to change the remote, run:

```
git remote remove origin
git remote add origin https://github.com/your-username/c-calculator.git
```

Why This Step Is Important

Linking to a remote repository allows you to:

- Push your code to GitHub for others to see
- Pull updates made by collaborators
- Back up your work online
- Contribute to group projects and open source

Once the remote is set, you're ready to push your code using `git push`.

3.7.8 Push Local Code to GitHub

After committing your changes locally and linking your repository to a remote on GitHub, the next step is to **push** your code online. This makes your project available on GitHub and allows others (like teammates or instructors) to view or collaborate on your work.

Syntax

For the first push:

```
git push -u origin main
```

For subsequent pushes:

```
git push
```

- `origin` refers to the remote repository name (set earlier).
- `main` refers to the branch you're pushing (the default main branch).
- The `-u` (or `--set-upstream`) option tells Git to remember this remote and branch so you can simply type `git push` next time.

Example

Continuing from the C project example:

```
git push -u origin main
```

If successful, Git will upload all committed changes to your GitHub repository. You'll see output confirming the push and the URL to your project, like:

```
Enumerating objects: 7, done.
Counting objects: 100% (7/7), done.
Writing objects: 100% (7/7), 2.13 KiB | 2.13 MiB/s, done.
To https://github.com/your-username/c-calculator.git
 * [new branch]      main -> main
```

First-Time Authentication (Important)

The first time you push to GitHub, it may prompt for authentication. GitHub no longer allows password-based login from the command line. Instead, use one of the following:

- **Personal Access Token (PAT)** – A secure key you generate on GitHub.
- **SSH key** – A more advanced, key-based authentication method.

Recommended for beginners: Personal Access Token

(a) Go to `https://github.com/settings/tokens`

(b) Click `Generate new token`

(c) Choose scopes like `repo`

(d) Copy the token and use it as your password when Git asks

Tip: Use a Git GUI like GitHub Desktop to avoid typing tokens manually.

How to Check if Push Worked

Go to your GitHub repository page (e.g., `https://github.com/your-username/c-calculator`) — you should see your files and commit message.

Common Push Errors and Fixes

- **Error: `failed to push some refs`** – This happens when your local branch is behind the remote. For example, if two group members are working on a single branch and one of the student has pushed something on the branch whereas the other has not downloaded (pull, we will see it shortly) the chages yet. In this case, we need to push the changes again but before that, we must resolve the conflicts i.e. accept the changes from commit.
- **Error: authentication failed** – Double-check your personal access token or SSH setup.
- **Branch mismatch** – If GitHub created a `main` branch but your local is still `master`, rename it:

```
    git branch -M main

```

Summary

Pushing your code uploads your latest commits to GitHub. You must first add and commit locally, then use `git push` to share changes.

3.7.9 Fetch Changes from Remote

When working collaboratively, other team members might push updates to the remote repository on GitHub. To get these updates without immediately applying them to your current work, you use the `git fetch` command.

What Does `git fetch` Do?

- Downloads new commits, branches, and tags from the remote repository
- Updates your local copy of the remote branches (e.g., `origin/main`)
- Does **not** change your current working files or branches
- Lets you review incoming changes before deciding to integrate them

Usage

```
git fetch origin
```

Here, `origin` is the name of the remote repository (default name for GitHub repos).

Example

Suppose your teammate added a new feature to the `main` branch on GitHub. To update your local repository with their changes, run:

```
git fetch origin
```

This command downloads their changes and updates your local tracking branch `origin/main`.

Check What Changed

To compare your local branch with the remote-tracking branch, use:

```
git diff main origin/main
```

This shows what files and lines differ between your current work and the remote version.

3.7.10 Pull Changes from Remote

While `git fetch` only downloads updates, `git pull` downloads **and** immediately merges those changes into your current branch. It is a convenient way to update your work with the latest changes from the remote repository.

What Does `git pull` Do?

- Runs `git fetch` to get the latest commits
- Automatically merges the fetched changes into your current branch
- Updates your working directory with the new code

Usage

```
git pull origin main
```

This fetches changes from the remote `main` branch and merges them into your local `main` branch.

Example

Continuing with the C project, if your teammate pushed bug fixes to the `main` branch on GitHub, you can update your local code and incorporate those fixes by running:

```
git pull origin main
```

Git will automatically merge the changes. If there are conflicts, Git will notify you, and you'll need to resolve them manually.

When to Use `git fetch` vs `git pull`

- Use `git fetch` if you want to review incoming changes before merging.
- Use `git pull` if you want to quickly update your local branch and working files.

Summary

When fetching from remote, merge conflicts can arrive which could be tricky for beginners to fix. However, `git fetch` and `git pull` do roughly the same task but are designed in a way to help the user to avoid the merge conflicts. `git fetch` is a safe way and gives you on overview of the scale of changes and conflicts before you decide to merge them locally using `git fetch`. In contrast, `git pull` is a faster way of merging the changes from remote branch in to you local branch but it could end in merge conflict which needs to be fixed manually. `git pull` tries to resolve maximum conflicts automatically and is a powerful tool, however as a beginner, it is suggested to used `git fetch`. Once you get an idea of conflicts, migrate to `git pull` gradually.

Command	**Action**	**Effect on Local Repo**
`git fetch`	Download updates from remote	Updates remote-tracking branches, does NOT change current files or branches
`git pull`	Download updates AND merge into current branch	Updates remote-tracking branches and merges changes into your working files

Using these commands effectively helps you keep your local repository in sync with the shared project on GitHub.

3.7.11 Checking Differences or Changes made in the Files

During development, it is important to see what changes you've made to your code. The `git diff` command lets you view the exact line-by-line differences between various versions of your files.

What Does `git diff` Show?

- Which lines were added, removed, or modified
- Changes not yet staged (by default)

- Differences between branches or commits
- Changes between local and remote repositories

Common Usages

(a) **View unstaged changes (working directory vs staging area):**

```
    git diff

```

(b) **View staged changes (staging area vs last commit):**

```
    git diff --cached

```

(c) **Compare local branch to remote branch:**

```
    git diff main origin/main

```

Example: C Project Context

Imagine you edited the following function in `math.c`:

```c
// Old version
int add(int a, int b) {
    return a + b;
}

// New version
int add(int a, int b) {
    printf("Adding␣%d␣and␣%d\n", a, b);
    return a + b;
}
```

Before committing the change, you can inspect it using:

```
git diff math.c
```

You'll see output like:

```
diff --git a/math.c b/math.c
index e69de29..a1b2c3d 100644
--- a/math.c
+++ b/math.c
@@ -1,3 +1,4 @@
+    printf("Adding␣%d␣and␣%d\n", a, b);
     return a + b;
```

This shows the line you added. Lines starting with '+' are additions, and lines starting with '-' are deletions.

Tips

- Use `git diff <filename>` to focus on one file
- Run `git diff --stat` for a summary of which files changed
- Use `git difftool` to open a visual comparison (if configured)

Why Use `git diff`?

- To review changes before committing
- To inspect changes pulled from teammates
- To debug or find where things went wrong

3.7.12 Create and Switch Branches

In Git, branches are like parallel versions of your project. You can use them to develop features, fix bugs, or try out experiments without affecting the main code. Once a branch is ready, you can merge it back into the main branch.

Why Use Branches?

- Keep feature development isolated from stable code
- Let team members work independently
- Avoid breaking the main project while testing a new feature
- Have a separate branch for debugging and testing purposes

Create and Switch to a New Branch

With newer versions of Git, the recommended command to create and switch to a branch is:

```
git switch -c feature-name
```

`-c` stands for "create." This creates the branch and switches to it in one step.

Example: C Project Feature Branch

Suppose you are adding a new scientific mode to your calculator project. You can create and switch to a new branch like this:

```
git switch -c scientific-mode
```

This does two things:

(a) Creates a new branch named `scientific-mode`

(b) Moves your working directory to that branch

Now, any changes you make will be isolated to this branch.

Switch Between Branches

To switch to an existing branch (for example, back to `main`), run:

```
git switch main
```

List All Branches

To see all local branches and check which one you're currently on:

```
git branch -v
```

The current branch will be marked with an asterisk (*).

Summary of Branch Commands

- `git switch -c new-branch` – Create and switch to a new branch
- `git switch existing-branch` – Switch to an existing branch
- `git branch` – List local branches

Branches allow multiple developers to work in parallel without interfering with each other. Once development is complete, you can merge the branch into `main` (covered next).

3.7.13 Merging Branches

After completing work in a feature branch, the next step is to integrate those changes back into the `main` branch (or another base branch). This is done using the `git merge` command.

What Does `git merge` Do?

- Combines the changes from one branch into another
- Preserves the full commit history
- Creates a merge commit if the branches have diverged

Steps to Merge a Feature Branch into Main

Let's say you've been working on a branch called `scientific-mode`, and now it's time to merge it into `main`.

Step 1: Switch to the branch you want to merge into

```
git switch main
```

Step 2: Merge the feature branch

```
git merge scientific-mode
```

Git will attempt to merge the changes automatically. If the two branches changed different parts of the project, Git will combine the changes and create a new merge commit.

Example: Merging a C Feature Branch

Suppose you implemented scientific calculator functions in `scientific-mode`. To include them in your main project:

```
git switch main
git merge scientific-mode
```

Git may respond with:

```
Updating abc123..def456
Fast-forward
 math.c | 20 ++++++++++++++++++++
 1 file changed, 20 insertions(+)
```

This means the merge was successful.

Handling Merge Conflicts (Expanded)

A **merge conflict** occurs when both branches have made changes to the same part of a file, and Git cannot decide which version to keep.

Example Conflict in `math.c`:

Imagine both branches changed the 'add()' function in different ways.

`main` branch:

```
int add(int a, int b) {
    return a + b;
}
```

`scientific-mode` branch:

```
int add(int a, int b) {
    printf("Adding␣%d␣and␣%d\n", a, b);
    return a + b;
}
```

When merging, Git will show a conflict like this in `math.c`:

```
int add(int a, int b) {
<<<<<<< HEAD
    return a + b;
=======
    printf("Adding␣%d␣and␣%d\n", a, b);
    return a + b;
>>>>>>> scientific-mode
}
```

The lines between <<<<<<< `HEAD` and ======= come from your current branch (usually `main`). The lines between ======= and >>>>>>> `branch-name` come from the branch you are merging (`scientific-mode` in this case).

How to Resolve the Conflict:

(a) Open the conflicted file(s) in a code editor.

(b) Manually edit the file to keep the desired version. For example:

```
// Final resolved version
int add(int a, int b) {
    printf("Adding␣%d␣and␣%d\n", a, b);
    return a + b;
}
```

(c) Remove all the conflict markers (<<<<<<<, =======, >>>>>>>).

(d) Save the file.

(e) Stage the resolved file:

```
git add math.c
```

(f) Finalize the merge with a commit:

```
git commit
```

Git will open a default merge commit message. You can leave it as is or write your own message describing the merge resolution.

Best Practices for Avoiding Conflicts

- Pull or fetch the latest changes from the remote repository before starting new work:

```
git pull origin main
```

- Communicate with teammates about which files each person is editing.
- Break large changes into smaller, isolated commits and branches.
- Review changes using `git diff` before merging.

Summary

- `git switch main` – Move to the branch you want to merge into.
- `git merge branch-name` – Combine the changes.
- If conflicts occur, manually edit the files and complete the merge with `git add` and `git commit`.

Merge conflicts are a normal part of collaboration. With practice, resolving them becomes straightforward and helps maintain clean, reliable code.

3.7.14 Optional: Remove a Tracked File

Sometimes you may want to remove a file from your Git repository — for example, if it was accidentally added, or if it's no longer needed in the project. Git provides commands to remove tracked files cleanly.

Remove and Delete the File from Disk

The simplest way to remove a file from both Git and your working directory is:

```
git rm filename
```

This will:

- Delete the file from your working directory
- Stage the deletion for the next commit

Example:

If you want to remove a temporary test file named `temp.c`, run:

```
git rm temp.c
git commit -m "Remove unused temp.c file"
```

Remove from Git but Keep the File on Disk

If you want to stop tracking a file (for example, a log or binary file), but keep the file itself in your local folder:

```
git rm --cached filename
```

Example:

To stop tracking a build output file named `output.log`, but leave it on disk:

```
git rm --cached output.log
git commit -m "Stop␣tracking␣output.log"
```

You can then add the file to your `.gitignore` to prevent Git from tracking it again in the future.

Common Use Cases

- Accidentally committed large or private files
- Remove auto-generated files (e.g., compiled binaries)
- Clean up outdated test or debug files

Summary

- `git rm file` — removes the file from Git and deletes it from disk
- `git rm --cached file` — removes the file from Git but keeps it locally
- Always commit the change after using `git rm`

This command helps keep your repository clean and focused on the files that matter.

Recommended Git Workflow

- `git status` – Check the status of changes.
- `git add <file>` – Stage specific files.
- `git commit -m "message"` – Save a snapshot.
- `git fetch/pull` – Sync with the latest version from GitHub.
- `git push` – Upload your commits to GitHub.

3.7.15 Summary of Essential Git Commands

The table below summarizes the key Git commands covered in this guide, arranged in the typical order of usage when working on a project. These commands form the foundation for effective version control and collaboration.

Command	Description
`git init`	Initialize a new Git repository in the current directory
`git status`	Show the current state of the working directory and staging area
`git add <file>`	Stage a file to be included in the next commit
`git rm <file>`	Remove a tracked file and delete it from disk
`git rm --cached <file>`	Stop tracking a file without deleting it locally
`git commit -m "message"`	Record the staged changes in the repository with a message
`git remote add origin` <URL>	Link a local repository to a remote GitHub repository
`git push -u origin main`	Push local commits to the remote `main` branch
`git fetch`	Download new data (commits, branches) from the remote repository
`git pull`	Fetch and automatically merge changes from the remote
`git diff`	Show differences between files (unstaged by default)
`git diff --cached`	Show staged changes ready to be committed
`git switch -c <branch>`	Create and switch to a new branch
`git switch <branch>`	Switch to an existing branch
`git branch`	List all local branches
`git merge <branch>`	Merge another branch into the current branch
`git log`	Show a history of commits

This table can be printed and kept as a handy reference while working on collaborative coding assignments or projects.

Further Learning Resources

- Interactive Git tutorial: `https://learngitbranching.js.org`
- GitHub's beginner guide: `https://docs.github.com/en/get-started/quickstart`
- Video Tutorial: `https://www.youtube.com/watch?v=RGOj5yH7evk`

Lab 1: Getting Started with RISC-V (Assembly Language) in VS Code

Objectives

This lab introduces the development workflow and tools required for the Computer Architecture laboratory. Students will install and configure Git, GitHub, and Visual Studio Code, practice essential version control operations, and become familiar with the RISC-V instruction set through basic assembly programming exercises involving arithmetic operations.

Background

Version control is essential for collaborative development, allowing multiple contributors to manage and track code changes. Git is the most widely used tool for version control, and GitHub provides a cloud platform to host and share repositories.
Repository is a data structure that stores metadata for a file(s) within a folder. GitHub is a server platform providing free account creation and repository storage.
In the first part of this lab focuses on software setup: installing Git and VS Code , then performing a complete Git workflow (clone, edit, commit, push).
The second part introduces RISC-V Assembly Language, an open and simplified instruction set architecture widely used for teaching computer architecture. we will start writing and managing RISC-V programs using VS Code.

Equipment and Software Required

- **Hardware:** A PC
- **Software:**
  - **Git** (version control)
  - **Visual Studio Code**
- **Accounts:**
  - A GitHub account

Task 1: Git and GitHub Workflow

Follow the steps below to complete the git setup workflow using Git and GitHub. This assumes that you are working on a Windows machine.

1. **Install Git**

   Download Git from: https://git-scm.com/download/win
2. **Verify Git Installation**

- Open **Git Bash**.
- Type the command:

```
git --version
```

- You should see output similar to: `git version 2.42.0.windows.1`

3. **Configure Git (Name and Email)**

   Open Git Bash and run the following (replacing with your actual name and email):

```
git config --global user.name "Your Full Name"
git config --global user.email "youremail@example.com"

```

   Verify with:

```
git config --global --list

```

4. **Create a GitHub Account**

   Log into your GitHub account using your user name/email and password.
   If you don't already have a GitHub account,

   - Go to `https://github.com`
   - Click on **Sign up** and create your GitHub account.
   - After signing up, log into your account.

5. **Create a New Repository on GitHub**

   – Click the + icon in the top right corner → `New Repository`
   – Name it: `Lab01-Git-YourName`
   – Leave ”`README`” and ”`.gitignore`” **unchecked**
   – Click **Create Repository**
   – Copy the **HTTPS URL**, e.g.: https://github.com/YourUsername/Lab01-Git-YourName.git

6. **Clone the Repository locally**

   – Navigate to the desired folder (e.g., `Documents\GitProjects`)
   – Right-click inside the folder and select `Git Bash Here`
   – In Git Bash, run:

```
git clone https://github.com/YourUsername/Lab01-Git-YourName.git
cd Lab01-Git-YourName

```

7. A folder `Lab01-Git-YourName` will appear.

Navigate to the Cloned Repository and Create, Commit a File

In the same Git Bash,

1. Navigate to the file:

```
cd Lab01-Git-YourName
```

2. Create a file:

```
touch hello.txt
```

3. Edit the file using nano:

```
nano bye.txt
```

In nano:

- Type your message: `This is my first Git lab file!`
- Press `Ctrl+O` to write, then `Enter` to confirm/save.
- Press `Ctrl+X` to exit.

4. Add, commit, and push:

```
git status
git add bye.txt
git commit -m "Added␣bye.txt"
git push origin main
```

Note: If `main` gives an error, try `git push origin master`.

5. **Verify on GitHub**

- Go to `https://github.com`

- Open your repository.
- You should see `bye.txt` committed.

For more on version control, refer to Chapter 3.

Tips

- **How to Paste in Git Bash:** Right-click and choose 'paste' option (`Ctrl+V` do not work).
- **To fix wrong email:** Re-run configuration command:

```
git config --global user.email "correct@email.com"

```

- **To fix wrong name:** Re-run configuration command:

```
git config --global user.name "correct␣name"

```

Task 02: Setting Up VS Code (RISC-V Simulation Environment)

To learn how RISC-V programs are written, executed, and tested, we'll be using Visual Studio Code (VS Code). This setup enables us to write and run RISC-V programs locally on our own computers—without relying on physical hardware or online simulators.
VS Code will be our primary environment for writing RISC-V assembly code. It's a lightweight and user-friendly editor that supports a wide range of extensions. We'll write our programs in .s files, which are simple text files containing RISC-V assembly instructions.
Unlike web-based tools like Venus that run in a browser, this offline setup provides a more robust and flexible development experience. It allows for greater control, supports larger projects, and integrates easily with other tools and workflows.
In this section, we will set up a RISC-V simulation environment using Visual Studio Code . This environment will allow you to write, run, and debug RISC-V assembly programs easily on your local machine.
Download and install from: `https://code.visualstudio.com`

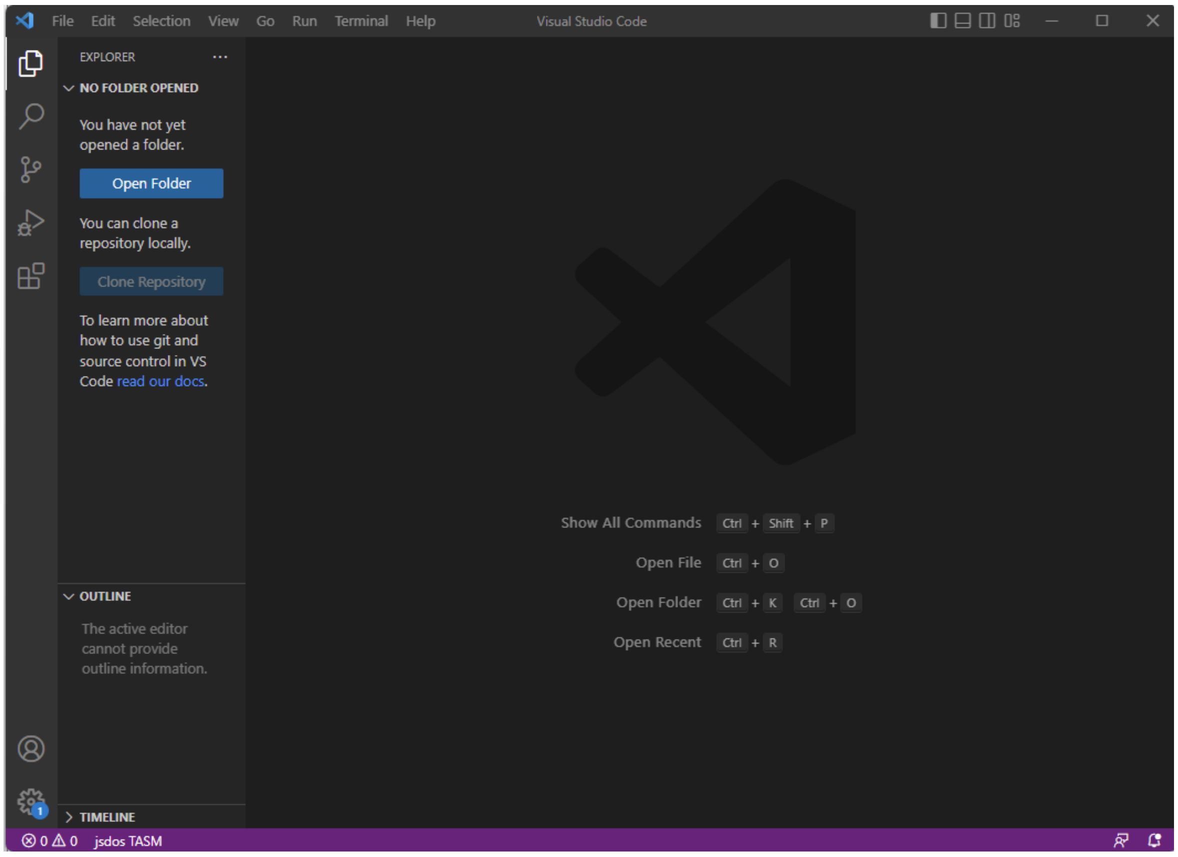
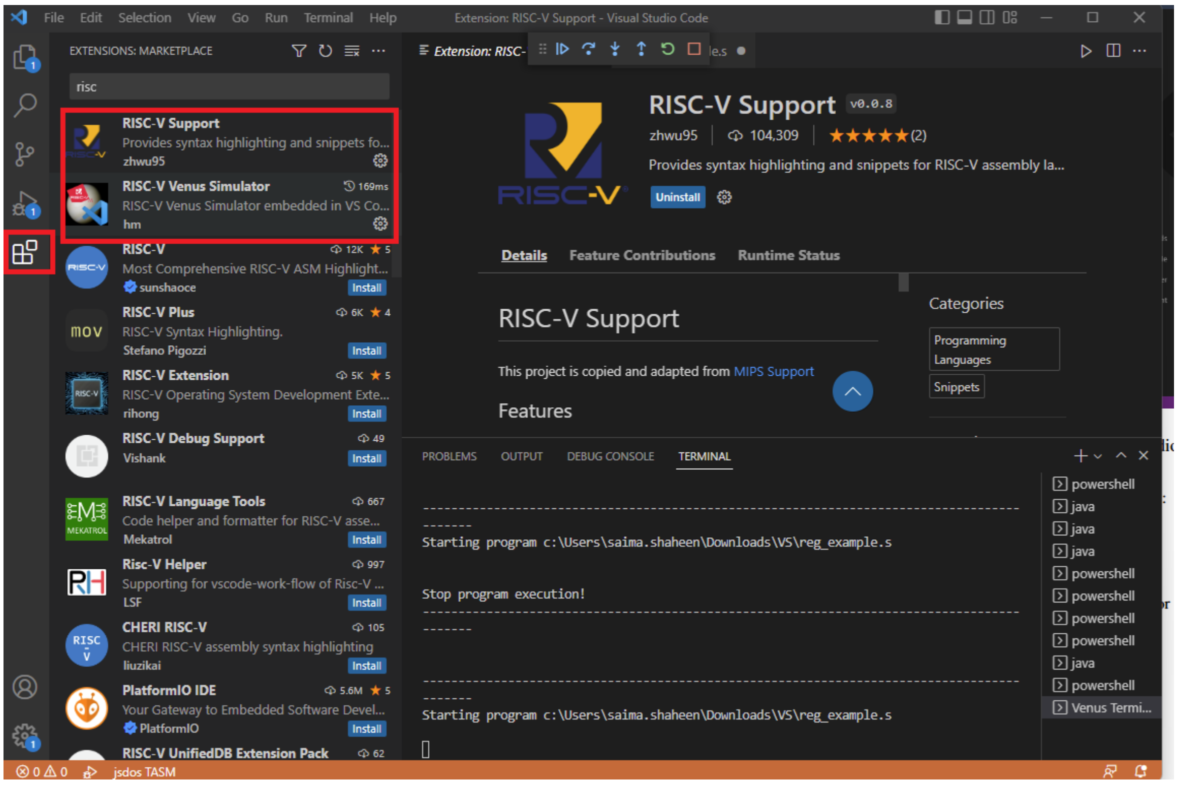

File
Edit
Selection
View
Go
Run
Terminal
Help
Visual Studio Code
EXPLORER
NO FOLDER OPENED
You have not yet opened a folder.
Open Folder
You can clone a repository locally.
Clone Repository
To learn more about how to use git and source control in VS Code read our docs.
OUTLINE
The active editor cannot provide outline information.
TIMELINE
Show All Commands Ctrl + Shift + P
Open File Ctrl + O
Open Folder Ctrl + K Ctrl + O
Open Recent Ctrl + R
0 0 jsdos TASM

Figure 1.1: Visual Studio Code

Open VS Code, Click on Left side Extension and install:

- Code Runner (for running assembly files)
- Assembly `x86` and `x86_64` syntax (optional for color coding)
- RISC-V Support
- RISC-V Venus Simulator



RISC-V Support
Provides syntax highlighting and snippets fo...
zhwu95

RISC-V Venus Simulator 169ms
RISC-V Venus Simulator embedded in VS Co...
hm

RISC-V 12K ★ 5
Most Comprehensive RISC-V ASM Highlight...
sunshaoce

RISC-V Plus 6K ★ 4
RISC-V Syntax Highlighting.
Stefano Pigozzi

RISC-V Extension 5K ★ 5
RISC-V Operating System Development Exte...
rihong

RISC-V Debug Support 49
Vishank

RISC-V Language Tools 667
Code helper and formatter for RISC-V asse...
Mekatrol

Risc-V Helper 997
Supporting for vscode-work-flow of Risc-V ...
LSF

CHERI RISC-V 105
CHERI RISC-V assembly syntax highlighting
liuzikai

PlatformIO IDE 5.6M ★ 5
Your Gateway to Embedded Software Devel...
PlatformIO

RISC-V UnifiedDB Extension Pack 62

RISC-V Support v0.0.8
zhwu95 | 104,309 | ★★★★★(2)
Provides syntax highlighting and snippets for RISC-V assembly la...

RISC-V Support
This project is copied and adapted from MIPS Support
Features

```
--------------------------------------------------------------------------------
-------
Starting program c:\Users\saima.shaheen\Downloads\VS\reg_example.s


Stop program execution!
--------------------------------------------------------------------------------
-------



--------------------------------------------------------------------------------
-------
Starting program c:\Users\saima.shaheen\Downloads\VS\reg_example.s

▯
```

Figure 1.2: Visual Studio Code

Functional Verification

Once your environment is properly set up , you can begin verifying that your RISC-V assembly code runs as expected. Visual Studio Code gives you powerful features to write, run, debug, and verify your code effectively.

Code View (Source Window)

The main area in VS Code displays your .s assembly file.

Original Assembly Code

You write your RISC-V assembly using either standard register names (e.g., x0, x1, etc.) or aliases like zero, sp, t0, etc. VS Code recognizes both and provides syntax highlighting if the RISC-V extension is installed.

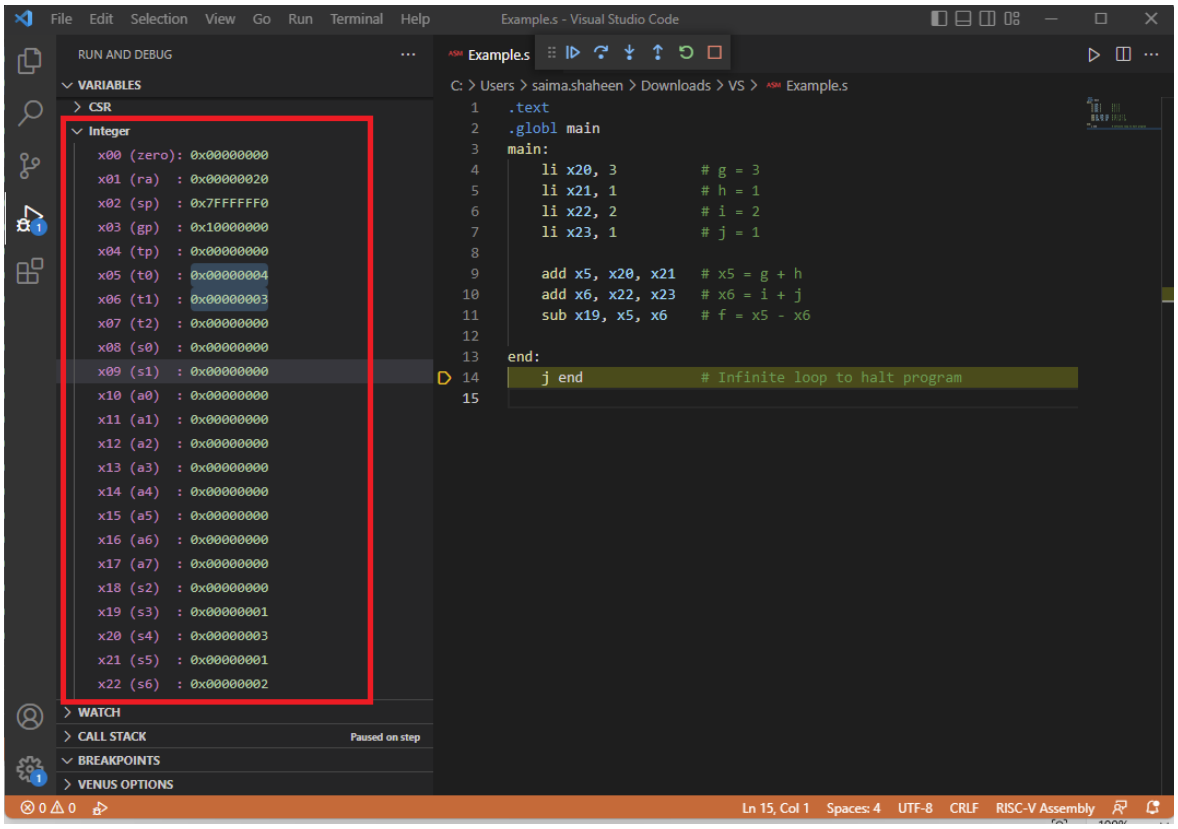
Breakpoint

You can click next to the line number to set breakpoints, which pause execution during debugging. This helps step through your code line-by-line.

Registers/Memory Window

Registers

This is where the simulator shows us the internal processor state that it is maintaining, i.e., the registers. All 32 RISC-V registers, from x0-x31, are visible here each showing the value it contains.


File Edit Selection View Go Run Terminal Help
Example.s - Visual Studio Code
RUN AND DEBUG
VARIABLES
CSR
Integer
Example.s
C: > Users > saima.shaheen > Downloads > VS > Example.s
WATCH
CALL STACK Paused on step
BREAKPOINTS
VENUS OPTIONS
Ln 15, Col 1 Spaces: 4 UTF-8 CRLF RISC-V Assembly

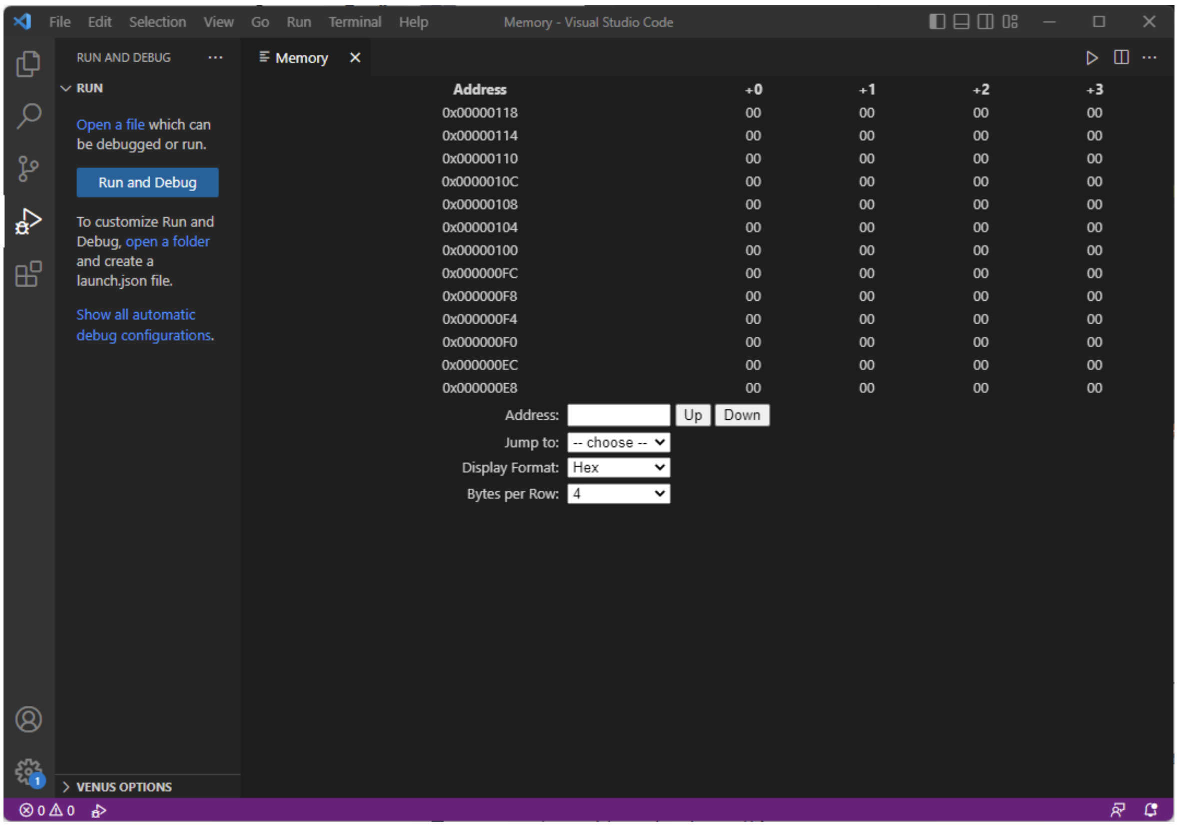
```
x00 (zero): 0x00000000
x01 (ra)  : 0x00000020
x02 (sp)  : 0x7FFFFFF0
x03 (gp)  : 0x10000000
x04 (tp)  : 0x00000000
x05 (t0)  : 0x00000004
x06 (t1)  : 0x00000003
x07 (t2)  : 0x00000000
x08 (s0)  : 0x00000000
x09 (s1)  : 0x00000000
x10 (a0)  : 0x00000000
x11 (a1)  : 0x00000000
x12 (a2)  : 0x00000000
x13 (a3)  : 0x00000000
x14 (a4)  : 0x00000000
x15 (a5)  : 0x00000000
x16 (a6)  : 0x00000000
x17 (a7)  : 0x00000000
x18 (s2)  : 0x00000000
x19 (s3)  : 0x00000001
x20 (s4)  : 0x00000003
x21 (s5)  : 0x00000001
x22 (s6)  : 0x00000002
```

```
.text
.globl main
main:
    li x20, 3          # g = 3
    li x21, 1          # h = 1
    li x22, 2          # i = 2
    li x23, 1          # j = 1

    add x5, x20, x21   # x5 = g + h
    add x6, x22, x23   # x6 = i + j
    sub x19, x5, x6    # f = x5 - x6

end:
    j end              # Infinite loop to halt program
```

Figure 1.3: RISC-V Registers

Memory

Now press Ctrl + Shift + P and select ”VENUS: Open Memory Viewer”
This shows us a view of memory that the simulator is maintaining. Memory as we know is made up of bytes and each byte has an address. We observe on the left “Address” column the addresses starting from 0x00 at the bottom and increasing progressively upwards.
To the right we have the memory locations pointed to by these addresses and the values contained in those memory locations. Each location is a byte and the column title +0, +1, +2, +3 gives the offset of that particular memory byte from the address shown in the right most column. Using the Address column with these offsets you can locate each byte of memory and the value it contains.



RUN AND DEBUG

RUN

Open a file which can be debugged or run.

To customize Run and Debug, open a folder and create a launch.json file.

Show all automatic debug configurations.

VENUS OPTIONS

Address	+0	+1	+2	+3
0x00000118	00	00	00	00
0x00000114	00	00	00	00
0x00000110	00	00	00	00
0x0000010C	00	00	00	00
0x00000108	00	00	00	00
0x00000104	00	00	00	00
0x00000100	00	00	00	00
0x000000FC	00	00	00	00
0x000000F8	00	00	00	00
0x000000F4	00	00	00	00
0x000000F0	00	00	00	00
0x000000EC	00	00	00	00
0x000000E8	00	00	00	00

Address:

Jump to: -- choose --

Display Format: Hex

Bytes per Row: 4

Figure 1.4: RISC-V Memory

Getting Started with RISC-V Programming

As you know assembly is a low-level language used to write programs for processors. Being a low-level language, it is hardware dependent which means, that every processor, architecture actually, has its own assembly language with its own syntax. What this means in practice is that Intel x86 assembly is different from ARM assembly which, in turn is different from POWER assembly, etc. RISC-V therefore has its own assembly language.

Assembler is a software that transforms this human readable assembly language into the machine language (binary) that the processor understands and can execute. In this lab we will write some very basic code in RISC-V assembly language, convert it into binary and try to execute it on a RISC-V simulator

What is Processor Assembly

Usually you need a processor, the hardware chip, to execute a program binary but there often there arise situations, like ours now, where access to the hardware processor is unavailable. To work around such situations, there exist software that mimic the behavior of a hardware processor. Such software(s) are called processor simulators.

Processor simulators behave exactly like the processors they simulate. They take the binary program as input, take the instructions in that program and execute them one by one. They maintain the internal state of a processor, i.e., its registers, it's pipeline and other circuits, in software and update this state in the light of the instructions being executed exactly as it would be updated by a real hardware processor chip.

Instruction Set Table

The table below shows the instruction set for RISC V.

Category	**Instruction**	**Example**	**Meaning**	**Comments**
Arithmetic	Add	`add x5, x6, x7`	$x5 = x6 + x7$	Three register operands; add
	Subtract	`sub x5, x6, x7`	$x5 = x6 - x7$	Three register operands; subtract
	Add immediate	`addi x5, x6, 20`	$x5 = x6 + 20$	Used to add constants
Data Transfer	Load doubleword	`ld x5, 40(x6)`	$x5 = \text{Memory}[x6 + 40]$	Doubleword from memory to register
	Store doubleword	`sd x5, 40(x6)`	$\text{Memory}[x6 + 40] = x5$	Doubleword from register to memory
	Load word	`lw x5, 40(x6)`	$x5 = \text{Memory}[x6 + 40]$	Word from memory to register
	Load word (unsigned)	`lwu x5, 40(x6)`	$x5 = \text{Memory}[x6 + 40]$	Unsigned word from memory to register
	Store word	`sw x5, 40(x6)`	$\text{Memory}[x6 + 40] = x5$	Word from register to memory
	Load halfword	`lh x5, 40(x6)`	$x5 = \text{Memory}[x6 + 40]$	Halfword from memory to register
	Load halfword (unsigned)	`lhu x5, 40(x6)`	$x5 = \text{Memory}[x6 + 40]$	Unsigned halfword from memory to register
	Store halfword	`sh x5, 40(x6)`	$\text{Memory}[x6 + 40] = x5$	Halfword from register to memory
	Load byte	`lb x5, 40(x6)`	$x5 = \text{Memory}[x6 + 40]$	Byte from memory to register
	Load byte (unsigned)	`lbu x5, 40(x6)`	$x5 = \text{Memory}[x6 + 40]$	Unsigned byte from memory to register
	Store byte	`sb x5, 40(x6)`	$\text{Memory}[x6 + 40] = x5$	Byte from register to memory
	Load reserved	`lr.d x5, (x6)`	$x5 = \text{Memory}[x6]$	Load; first half of atomic swap
	Store conditional	`sc.d x7, x5, (x6)`	$\text{Memory}[x6] = x5$ or $0/1$	Store; second half of atomic swap
(Other)	Load upper immediate	`lui x5, 0x12345`	$x5 = 0x12345000$	Loads 20-bit constant shifted left 12 bits

Table 3.1: Summary of Common RISC-V Instructions

Using Arithmetic

In this task we will learn the use of RISC-V arithmetic instructions.

Example 1

The following example will help us to understand the basics of registers and RISC-V It is the compiler's job to associate program variables with registers. Consider the following assignment statement from an earlier example:

$$f = (g + h) - (i + j) \tag{3.1}$$

Key Terms

Word: A natural unit of access in a computer, usually a group of 32 bits.

Double Word: A natural unit of access in a computer, usually a group of 64 bits; corresponds to the size of a register in the RISC-V architecture.

Example: Assume that the variables f, g, h, i, and j are assigned to registers $x19, x20, x21, x22$, and $x23$, respectively. What is the corresponding RISC-V assembly code?

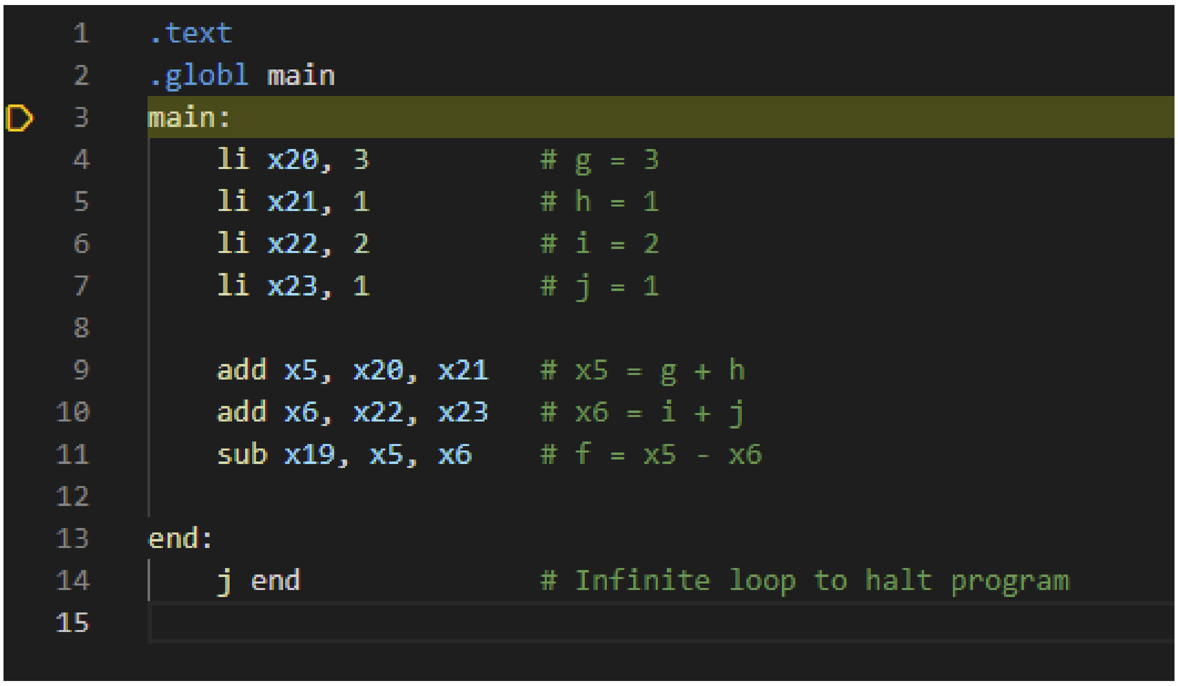
The compiled program replaces variables with their assigned registers and uses two temporary registers, $x5$ and $x6$, to store intermediate results:

```
add x5, x20, x21     // x5 = g + h
add x6, x22, x23     // x6 = i + j
sub x19, x5, x6      // f = (g + h) - (i + j)
```

Running Your First RISC-V Program in VS Code

Step 1: Create a New File

- Open Visual Studio Code.
- Press Ctrl + N to create a new file.
- Paste the following RISC-V assembly code into the editor:



```
.text
.globl main
main:
    li x20, 3           # g = 3
    li x21, 1           # h = 1
    li x22, 2           # i = 2
    li x23, 1           # j = 1

    add x5, x20, x21    # x5 = g + h
    add x6, x22, x23    # x6 = i + j
    sub x19, x5, x6     # f = x5 - x6

end:
    j end               # Infinite loop to halt program
```

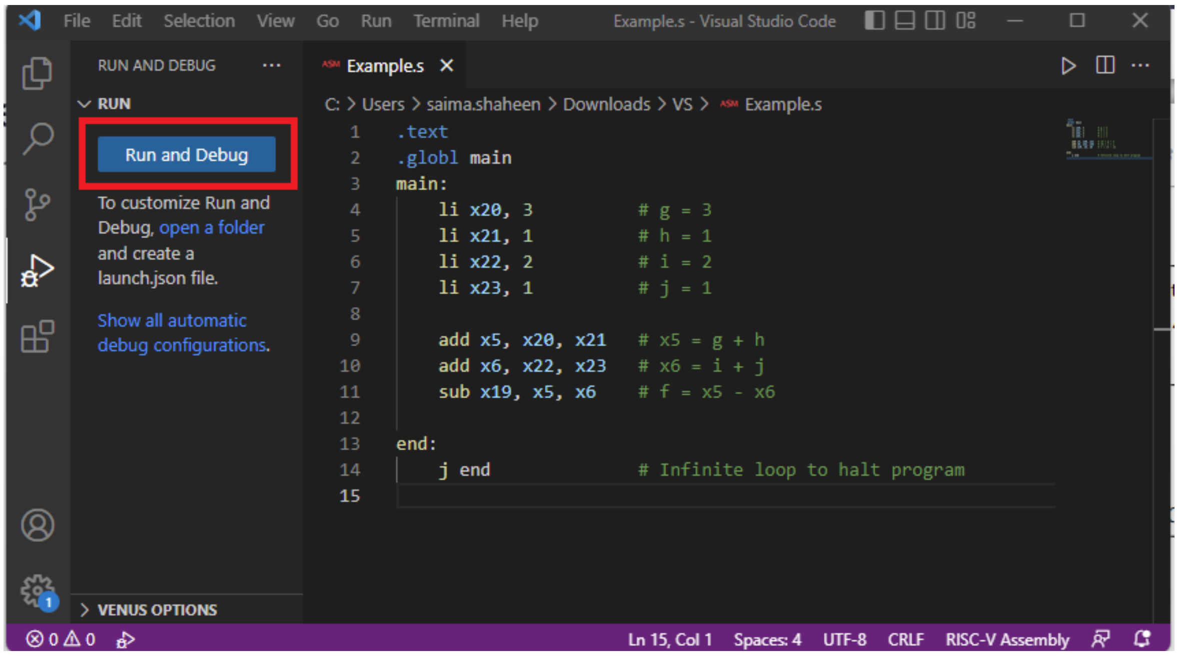
Figure 1.5: Assembly Code

Step 2: Save the File

- Press Ctrl + S.
- Save the file with a .s extension, e.g., Example.s.

Step 3: Run and Debug

To run and debug the program, press Ctrl + Shift + D or click on the ”Run and Debug” icon in the left sidebar.


File Edit Selection View Go Run Terminal Help
Example.s - Visual Studio Code
RUN AND DEBUG
RUN
Run and Debug
To customize Run and Debug, open a folder and create a launch.json file.
Show all automatic debug configurations.
VENUS OPTIONS
Example.s
C: > Users > saima.shaheen > Downloads > VS > Example.s
Ln 15, Col 1 Spaces: 4 UTF-8 CRLF RISC-V Assembly

```
.text
.globl main
main:
    li x20, 3        # g = 3
    li x21, 1        # h = 1
    li x22, 2        # i = 2
    li x23, 1        # j = 1

    add x5, x20, x21   # x5 = g + h
    add x6, x22, x23   # x6 = i + j
    sub x19, x5, x6    # f = x5 - x6

end:
    j end              # Infinite loop to halt program
```

Figure 1.6: Run and Debug

As you launch the debugger, you could see a control panel on top like:

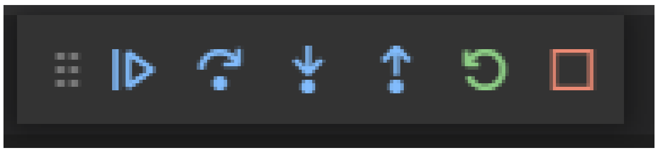


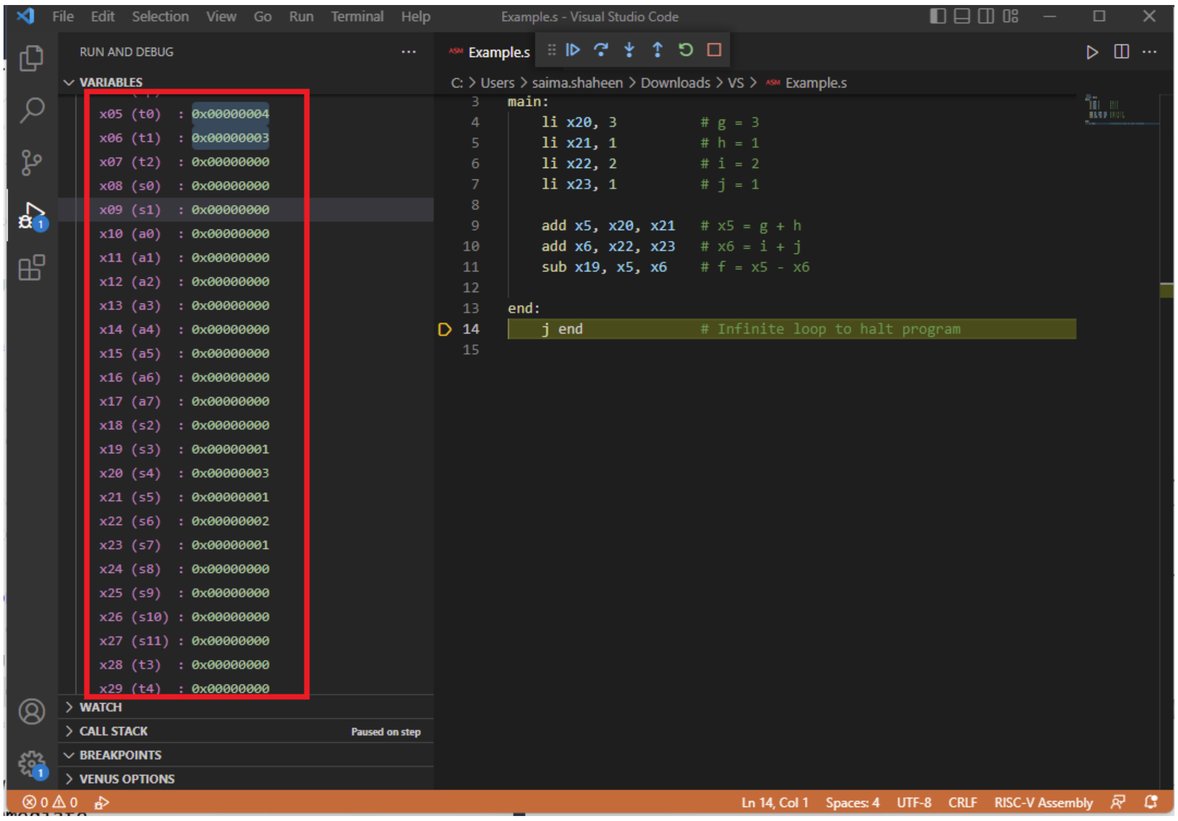
Figure 2.7: Control panel

From left to right, for each button:

- Continue to next breakpoint
- Step over an instruction, will not enter subroutine
- Step into an instruction, if it is a function call, will enter the function
- Step out a function call
- Restart debugging
- Stop the debugging

Step 4: Viewing Register Values

You can check the values stored in the registers by looking at the Registers section located on the left side of the screen.



```
x05 (t0)  : 0x00000004
x06 (t1)  : 0x00000003
x07 (t2)  : 0x00000000
x08 (s0)  : 0x00000000
x09 (s1)  : 0x00000000
x10 (a0)  : 0x00000000
x11 (a1)  : 0x00000000
x12 (a2)  : 0x00000000
x13 (a3)  : 0x00000000
x14 (a4)  : 0x00000000
x15 (a5)  : 0x00000000
x16 (a6)  : 0x00000000
x17 (a7)  : 0x00000000
x18 (s2)  : 0x00000000
x19 (s3)  : 0x00000001
x20 (s4)  : 0x00000003
x21 (s5)  : 0x00000001
x22 (s6)  : 0x00000002
x23 (s7)  : 0x00000001
x24 (s8)  : 0x00000000
x25 (s9)  : 0x00000000
x26 (s10) : 0x00000000
x27 (s11) : 0x00000000
x28 (t3)  : 0x00000000
x29 (t4)  : 0x00000000
```

```
main:
    li x20, 3          # g = 3
    li x21, 1          # h = 1
    li x22, 2          # i = 2
    li x23, 1          # j = 1

    add x5, x20, x21   # x5 = g + h
    add x6, x22, x23   # x6 = i + j
    sub x19, x5, x6    # f = x5 - x6

end:
    j end              # Infinite loop to halt program
```

Figure 1.8: Register Values

Example 2

The following example will help us to understand the add immediate

```
This quick add instruction with one constant operand is called add immediate
    or addi. To add 4 to register x22, we just write
addi x22, x22, 4 // x22 = x22 + 4
```

Task 3

Convert the following statement to RISC V. You can use the same registers as given in

```
int a = 5;
int b = 0 + 0;
a = b + 32;
int d = (a + b) - 5;
int e = (((a - d) + (b - a)) + d);
e = a + b + d + e;
```

Using memory instructions

RISC-V provides you a variety of instructions to load data from RAM into the registers as well as to store data from the registers into the memory. You will choose an instruction based on the type (whether it is signed or unsigned) and the size (whether it is a char, short, int, or long) of your data. See the RISC-V green card for a summary of memory instructions.

Example 3

The following example will help us to understand the add immediate

Example: Array Access in RISC-V

Assume that the variable `h` is associated with register `x21`, and the base address of the array `A` is stored in register `x22`. Consider the following C assignment statement:

$$A[12] = h + A[8] \tag{3.2}$$

Although this statement represents a single operation in C, both operands `h` and `A[8]` must be accessed explicitly in RISC-V assembly. Since array elements are stored in memory, additional instructions are required.

The offset used in the load and store instructions is calculated using **byte addressing**. Assuming each element of `A` is a word (4 bytes):

- `A[8]` $\rightarrow$ offset $8 \times 4 = 32$ bytes
- `A[12]` $\rightarrow$ offset $12 \times 4 = 48$ bytes

The corresponding RISC-V assembly code is shown below:

```
lw   x9, 32(x22)     // x9 = A[8]
add  x9, x21, x9     // x9 = h + A[8]
sw   x9, 48(x22)     // A[12] = h + A[8]
```

Task - 4a

Initialize the register x10 and x11 with values 0x78786464, 0xA8A81919, respectively manually.
Write the RISC-V assembly code for each item below. Try guessing the result in each destination before executing the instruction and corroborate it after execution:

Store x10 as unsigned integer at address 0x100.

Store x11 as unsigned integer at address 0x1F0.

Load an unsigned short integer (two bytes) from address 0x100 in x12.

Load a short integer from address 0x1F0 in register x13.

Load a singed character from address 0x1F0 in register x14.

Task 4b – Loop unrolling

Problem Statement

Assume there are three arrays a, b, and c located at the following addresses:

- a at address 0x100
- b at address 0x200
- c at address 0x300

Consider the following C code:

```c
for (int i = 0; i < 4; i++)
    c[i] = a[i] + b[i];
```

Write equivalent RISC-V assembly code for the above C code.
You have not studied loops yet; however, the above code can be written without using loop instructions.
Assume:

- a is a character array
- b is a short array
- c is an unsigned integer array

Assessment Rubric
Lab 1: Getting Started with RISC-V (Assembly Language) in VS Code

Name:	**Student ID:**	**section*:**

Points Distribution

	Task No.	**LR 2 Code**	**LR 5 Results**
In - Lab	**Task 1**	/0	/15
	Task 2	/0	/15
	Task 3	/10	/5
	Task 4a	/10	/5
	Task 4b	/10	/10
Total Points: 100		**/30**	**/50**
CLO Mapped		CLO 2	

Affective Domain Rubric		**Points**	**CLO Mapped**
AR7	**Report Submission & Git Upload**	/10 & /10	CLO 2

CLO	**Total Points**	**Points Obtained**
2	100	
Total	**100**	

For description of different levels of the mapped rubrics, please refer to the Lab Evaluation Assessment Rubrics and Affective Domain Assessment Rubrics provided here.

Lab 2: Implementing Decision Instructions in RISC V Assembly Language

Objectives

This lab aims to develop students' understanding of RISC-V assembly language with a focus on control flow and branch instructions. Students will write, execute, and verify RISC-V assembly programs using Visual Studio Code, and will reinforce their understanding through guided exercises completed either in the lab or independently.

Pseudo Instructions

RISC-V registers are named x0 through x31. Except x0, which is hardwired to contain all zeros, the rest can be used to store arbitrary data. Still, a few conventions have been defined about the use of these registers and things run much smoother when everybody follows them. Table 1.1 provides a summary of conventional use of various registers.

RISC-V Registers			
Register	**ABI Name**	**Description**	**Saved by**
x0	zero	zero register (hardwired zero)	
x1	ra	return address	Caller
x2	sp	stack pointer	Callee
x3	gp	global pointer	
x4	tp	thread pointer	
x5–7	t0–2	temporary registers	Caller
x8	s0 / fp	saved register / frame pointer	Callee
x9	s1	saved register	Callee
x10–11	a0–1	function arguments / return values	Caller
x12–17	a2–7	function arguments	Caller
x18–27	s2–11	saved registers	Callee
x28–31	t3–6	temporary registers	Caller

Table 3.2: RISC-V Registers

The second column lists their pseudonyms which can be used in the assembly language as their original name equivalents. These pseudonyms are easier to remember and also give an inkling to the conventional use of the register. Don't worry about the last column (yet).

Similarly, for ease of use, the assembler provides a series of pseudo instructions which are easier to remember and are more user friendly. The assembler translates these pseudo instructions into machine code for RISC-V equivalent instructions. Some of the pseudo instructions and their equivalent real RISC-V assembly instructions can be found on page 7 of this PDF at: `https://compillyon.gitlabpages.inria.fr/compilyon/CAP1920_ENSL/riscv_isa.pdf`.

You can see from their syntax that some of them, i.e., nop, li, mv etc., can be quite handy.
Advice: One advice is to follow one coding style only i.e., either use the pseudo instructions or use the original instructions with the original register names. Mixing them up will almost surely result in bugs. Same goes for register names.

Decision Instructions

Decision instructions help us manipulate the control flow of a program. Normally the processor would load an instruction, from the address stored in PC, from the RAM and execute it. While executing it would update the value of PC by adding 4 to it, i.e. PC = PC+4; , so that it now points to the next instruction in RAM. This process is repeated every clock cycle and the processor will proceed to continue executing instructions in a linear fashion forever, that is unless it runs out of RAM.

If we want to change this linear control flow, i.e. we want to execute some instruction other than the one next in line, we need to load the address of that instruction into the PC so that it becomes the next instruction to be executed. How do we know the address of a particular instruction? The assembler comes to our rescue here with the concept of labels.

Category	Instruction	Assembly	Operation	Description
Conditional branch	Branch if equal	`beq x5, x6, 100`	if (x5 == x6) go to PC+100	PC-relative branch if registers equal
	Branch if not equal	`bne x5, x6, 100`	if ($x5 \neq x6$) go to PC+100	PC-relative branch if registers not equal
	Branch if less than	`blt x5, x6, 100`	if ($x5 < x6$) go to PC+100	PC-relative branch if registers less
	Branch if greater or equal	`bge x5, x6, 100`	if ($x5 \geq x6$) go to PC+100	PC-relative branch if registers greater or equal
	Branch if less, unsigned	`bltu x5, x6, 100`	if ($x5 < x6$) go to PC+100	PC-relative branch if registers less, unsigned
	Branch if greater or equal, unsigned	`bgeu x5, x6, 100`	if ($x5 \geq x6$) go to PC+100	PC-relative branch if registers greater or equal, unsigned
Unconditional branch	Jump and link	`jal x1, 100`	x1 = PC+4; go to PC+100	PC-relative procedure call
	Jump and link register	`jalr x1, 100(x5)`	x1 = PC+4; go to x5+100	Procedure return; indirect call

Table 3.3: RISC-V Conditional and Unconditional Branch Instructions

Labels

Labels are text strings, followed by a colon, that we can put before instructions. These strings can be used as a substitute for addresses of those instructions. Listing 1 shows the use of labels. In this piece of code, `label1` and `label2` can be used instead of the addresses of the instructions that follow them. We will see next how to use these labels

```
  add x2, x3, x4
label1:
  sub x3, x0, x2    # label1 contain this instruction address
  addi x3, x3, 1

label2: sub x5, x6, x3  # label2 contain this instruction address
```

Listing 1: Labels

Conditional Branches

Branches are instructions which can be used to jump to a particular instruction by supplying its label. If the branch instruction performs the jump based on a particular condition, then such branches are called conditional branches.

Two of the principal conditional branch instructions provided in the RISC-V ISA are called `beq` and `bne` as shown in Listing 2 where `L1` and `L2` both are labels as defined previously.

```
beq x22, x23, L1
bne x22, x23, L2
```

Listing 2: Labels

The instruction `beq` will jump to the instruction whose label is `L1` if the values in registers x22 and x23 are equal. That is to say that after the execution of this instruction the PC will contain the address of the instruction whose label is `L1` and the processor will load that instruction for execution next instead of the one after the `beq`instruction. However, if the values in registers x22 and x23 are not equal, the jump will not happen, and the PC will be updated with the value PC+4 and the processor will continue with the execution of the next instruction in RAM.

The instruction `bne` behaves opposite to `beq`, i.e., it will jump to `L2` if the values in the two registers are unequal and continue with the next instruction in RAM if they are not.

Listings 3 and 4 show how these instructions can be used to implement if-else statements and loops.

```
  if (i == j)
    f = g + h;
  else
    f = g - h;
  // code after if/else goes here



  //assuming that variables f to j are in registers x19-x23
  bne x22, x23, Else
  add x19, x20, x21
  beq x0, x0, Exit  //un conditional jump
Else: sub x19, x20, x21

Exit: # the code after if/else goes here
```

Listing 3: If statement

The following code looks up for an odd value in an array. Let's assume that 'k=2' and array save[i]=[2 2 4 2], the while loop runs until it detects any value other than '2', x10 gives you the address of odd value in the data.

```
while (save[i] == k)
  i += 1;


// assuming i and k in x22 and x24, and the base address of Save in x25
Loop: slli x10, x22, 3 // Temp reg x10 = i * 8
    add x10, x10, x25 // x10 = address of save[i]
    ld x9, 0(x10) // Temp reg x9 = save[i]
    bne x9, x24, Exit // go to Exit if save[i] != k
    addi x22, x22, 1 // i = i + 1
    beq x0, x0, Loop // go to Loop
Exit:

```

Listing 4: while loop

Task 1:

Type the codes in Listings 3 and 4 in the assembler editor and then in the simulator look at their binary:

- The beq and bne instructions are stored in the SB format with their 32-bit binary. Look it up and try to find out what values their different fields contain.
- Looking at the binary formats, find out how do the labels are stored in the 32-bit instruction. Compare the label values stored inside the instruction to the code and try to guess how the assembler stores the jump addresses inside the instruction. (Hint: it uses a PC-relative scheme) Refer to following figures (from book page 119) to see encoding of instructions.

  Note: To test listing 4, ld will be replaced by lw and for computation of address, offset should be obtained by i*4 i.e., shift left by 2 instead of

Task 2:

The switch statement is similar to if/else statements. Write the equivalent RISC-V assembly code for Listing 5.
Assume that the variables x, a, b & c are signed integers and stored in x20, x21, x22, x23 respectively. Make sure to first assign suitable values for b and c first.

```c
switch (x) {
  case 1:
    a = b+c;
    break;
  case 2:
    a = b-c;
    break;
  case 3:
    a = b * 2;
    break;
  case 4:
    a = b / 2;
    break;
  default:
    a = 0;
}
```

Listing 5: task 1b.c

Task 3:

Write the equivalent RISC-V assembly code for Listing 6. Assume that the variables i and sum are in x22 and x23, while the array a located at address 0x200 is of 4-byte integers.

```c
for(int i=0; i<10; i++)
  a[i] = i;

for(int i=0; i<10; i++)
  sum = sum+a[i];
```

Listing 6: Reduction Sum

Task 4 (Challenge):

Translate the following C code to RISC-V assembly code. Use a minimum number of instructions. Assume that the values of a, b, i, and j are in registers x5, x6, x7, and x29, respectively. Also, assume that register x10 holds the base address of the array D.

```c

for(i=0; i<a; i++)
  for(j=0; j<b; j++)
    D[4*j] = i + j;
```

Listing 7: Nested Loops

Assessment Rubric
Lab 2: Implementing Decision Instructions in RISC V Assembly Language

Name:	**Student ID:**	**Section:**

Points Distribution

	Task No.	LR 2 Code	LR 5 Results
In - Lab	Task 1	/0	/5
	Task 2	/10	/10
	Task 3	/15	/10
	Task 4	/20	/10
Total Points: 100		**/45**	**/35**
CLO Mapped		CLO2	

Affective Domain Rubric		Points	CLO Mapped
AR7	**Report Submission**	/20	CLO 2

CLO	Total Points	Points Obtained
2	100	
Total	**100**	

For description of different levels of the mapped rubrics, please refer to the Lab Evaluation Assessment Rubrics and Affective Domain Assessment Rubrics provided here.

Lab 3: Implementing Jump and Return Instructions in RISC V Assembly Language

Objectives

This lab introduces RISC-V assembly instructions related to control transfer, specifically jump and return operations. Students will implement and analyze jump-based program flow using Visual Studio Code by completing a series of hands-on tasks designed to strengthen their understanding of subroutines and control flow in RISC-V.

Introduction

In this lab we will learn about:

- RISC-V Jump and Return instructions
- Handle the stack

Function calls

Functions, or procedures as they are sometimes called, are an important part of programming. They help avoid repeating the same code all over our programs. A function call would take the program flow to the start of the function code. That code will be executed, and a return from-function instruction would take the program flow back to where originally the function was called from and continue the execution from the next instruction after the function call.

Like all instruction sets, RISC-V provides special instructions to enable function calls and return from functions.

Category	Instruction	Assembly	Operation	Description
Unconditional branch	Jump and link	`jal x1, 100`	x1 = PC+4; go to PC+100	PC-relative procedure call
	Jump and link register	`jalr x1, 100(x5)`	x1 = PC+4; go to x5+100	Procedure return; indirect call

Table 3.1: RISC-V Unconditional Branch Instructions

You can think of a procedure like a spy who leaves with a secret plan, acquires resources, performs the task, covers his or her tracks, and then returns to the point of origin with the desired result. Nothing else should be perturbed once the mission is complete. Moreover, a spy operates on only a “need to know” basis, so the spy can’t make assumptions about the spymaster. Similarly, in the execution of a procedure, the program must follow these six steps:

- Put parameters in a place where the procedure can access them.
- Transfer control to the procedure.

- Acquire the storage resources needed for the procedure.
- Perform the desired task.
- Put the result value in a place where the calling program can access it.
- Return control to the point of origin, since a procedure can be called from several points in a program

```
jal    x1, func_name
jalr   x0, 0(x1)
```

Listing 1: Function instructions

Listing 1 lists the jal and jalr instructions used for function calls and returns from functions respectively in RISC-V.

Executing jal instruction would make the program flow jump to func name label in the code while at the same time storing the address of the next instruction PC+4 in the register x1. The return address needs to be stored so that the program flow can go back to where the function call was made to continue execution from there onwards once it has finished executing the function code. Since a function can be called from multiple locations in a program, without saving the return address we will not have a clue as to where to return to. The jalr instruction is used to return from functions. The jump target address is provided in a register. The example shown in Listing 1, it will jump to the address stored in the register x1. Combining this with jal instruction we can see how the two instructions work in tandem to provide the functionality of calling a function and returning from it.

RISC-V Registers			
Register	**ABI Name**	**Description**	**Saved by**
x0	zero	zero register (hardwired zero)	
x1	ra	return address	Caller
x2	sp	stack pointer	Callee
x3	gp	global pointer	
x4	tp	thread pointer	
x5–7	t0–2	temporary registers	Caller
x8	s0 / fp	saved register / frame pointer	Callee
x9	s1	saved register	Callee
x10–11	a0–1	function arguments / return values	Caller
x12–17	a2–7	function arguments	Caller
x18–27	s2–11	saved registers	Callee
x28–31	t3–6	temporary registers	Caller

Table 3.2: RISC-V Registers

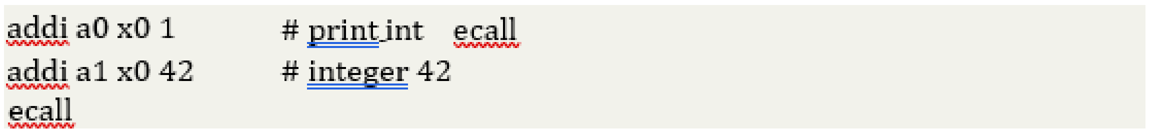
Another issue associated with function calls is passing the parameters and returning the result. As can be seen from Table 3.2, the RISC-V convention reserves the registers x10-x17 for passing parameters to a function when calling it. The caller code would fill in these registers the parameters needed by a function before calling it. Once the function is called, it will access its parameters from these registers, use them, and then return the result in x10 before calling the jalr instruction which would take the control flow to the caller who can then access the result of the function call by reading x10.

In case the registers x10-x17 contains some values, which will still be used by the caller code after the function call, the caller code needs to store these values as well as its own return address x1 on the stack before overwriting these values with the ones needed by the callee. See section 2.8 in the textbook.

Environment Calls

The Venus simulator supports some system calls, which it calls Environment Calls, as well. See the url `https://github.com/kvakil/venus/wiki/Environmental-Calls`. To execute an environment call we have to put its number in the register a0(x10), any argument it might need in a1(x11) and then call the ecall instruction. The simulator will perform the system call specified in the a0(x10) register.

The RISC-V environment calls can be used to print values from registers or memory in the Venus Terminal. For example, one of the supported system calls is print int whose number is 1 and which will print any value stored in the register a1(x11) in the terminal as an integer.



```
addi a0 x0 1        # print_int   ecall
addi a1 x0 42       # integer 42
ecall
```

Listing 2: Print integer

The code in listing 2, taken from the above url, would print the integer 42 on the Venus Terminal Window.

Task 1:

Write a code for the function sum shown in listing 3. It takes two integers a and b as arguments and returns their sum. You should also write code that calls this function i.e. it sets up the two arguments a and b in registers x10 and x11 and then calls sum(). After calling sum(), it then retrieves the returned result from the register x10. It can display the result using the ecall instruction.

```
int sum(int a, int b)
{
   return a+b;
}
```

```
addi x10,x0,12
addi x11,x0,12
jal x1,sum #calls the function named sum
addi x11,x10,0
li x10,1
ecall
j exit
sum:
   add x10,x11,x10
   jalr x0,0(x1)
exit:
```

Listing 3: Sum

Task 2:

Have a look at a C procedure:

```
long long int leaf_example (long long int g, long long int h, long long int i,
long long int j)
 {
        long long int f;
        f = (g + h) − (i + j);
        return f;
}
```

What is the RISC-V assembly code for the above procedure?

The parameter variables g, h, i, and j correspond to the argument registers x10, x11, x12, and x13and f corresponds to register x20. In addition, the program uses two temporary registers x18 and x19. Thus, we need to save three registers x18, x19, and x20 onto the stack.

Listing 4: Leaf Procedure

Task 3:

The following C procedure swaps two array elements (or two locations in memory)

```
    void swap (long long int v[ ], size_t k)

{

      long long int temp;
       temp = v[k];
       v[k] = v[k+1];
       v[k+1] = temp;
}
```

What is the RISC-V assembly code for the above procedure?

The parameter variables v[ ] and k correspond to the argument registers x10 and x11.

Listing 5: Swap Procedure

Task 4:

The procedure strcpy copies string y to string x using the null byte termination convention of C:

```
void strcpy (char x [], char y[])

{

size_t i;

i = 0;

while ((x[i] = y[i]) != '\0') /* copy & test byte */

i += 1;

}
```

What is the RISC-V assembly code?

Below is the basic RISC V assembly code segment, Assume that base addresses for arrays x and y are found in x10 and x11, while i is in x19, strcpy adjusts the stack pointer and then saves the saved register in x19 on the stack:

```
strcpy:

 addi sp, sp, -8          // adjust stack for 1 more item

 sd x19, 0(sp)            // save x19
```

Assessment Rubric
Lab 3: Implementing Jump and Return Instructions in RISC V Assembly Language

Name:	**Student ID:**	**Section:**

Points Distribution

	Task No.	LR 2 Code	LR 5 Results
In - Lab	**Task 1**	/10	/05
	Task 2	/10	/10
	Task 3	/10	/10
	Task 4	/15	/10
Total Points: 100		**/45**	**/35**
CLO Mapped		CLO 2	

Affective Domain Rubric		Points	CLO Mapped
AR7	**Report Submission & Git Upload**	/10 & /10	CLO 2

CLO	Total Points	Points Obtained
2	100	
Total	**100**	

For description of different levels of the mapped rubrics, please refer to the Lab Evaluation Assessment Rubrics and Affective Domain Assessment Rubrics provided here.

Lab 4: Implementing Nested Procedures and Sorting in RISC V Assembly Language

Objective

This lab focuses on the implementation of nested procedures in RISC-V assembly language. Students will design and analyze recursive and iterative procedures using Visual Studio Code, gaining practical understanding of procedure calls, stack usage, and control flow through structured hands-on tasks.

Nested Procedures

Procedures that do not call others are called leaf procedures. Just as a spy might employ other spies as part of a mission, who in turn might use even more spies, so do procedures invoke other procedures. Moreover, recursive procedures even invoke "clones" of themselves. Just as we need to be careful when using registers in procedures, attention must be paid when invoking non-leaf procedures.
For example, suppose that the main program calls procedure A with an argument of 3, by placing the value 3 into register x10 and then using jal x1, A. Then suppose that procedure A calls procedure B via jal x1, B with an argument of 7, also placed in x10. Since A hasn't finished its task yet, there is a conflict over the use of register x10. Similarly, there is a conflict over the return address in register x1, since it now has the return address for B. Unless we take steps to prevent the problem, this conflict will eliminate procedure A's ability to return to its caller.
One solution is to push all the other registers that must be preserved on the stack, just as we do with the saved registers. The caller pushes any argument registers (x10–x17) or temporary registers (x5-x7 and x28-x31) that are needed after the call. The callee pushes the return address register x1 and any saved registers (x8-x9 and x18-x27) used by the callee. The stack pointer sp is adjusted to account for the number of registers placed on the stack.Upon the return, the registers are restored from memory, and the stack pointer is readjusted.

Recursive Implementation

Recursive Implementation

Listing 3.1 shows a recursive C procedure that calculates the factorial of a number.

```c
long long int fact(long long int n) {
    if (n < 1)
        return 1;
    else
        return n * fact(n - 1);
}
```

Listing 3.1: Recursive factorial function in C

The RISC-V assembly implementation corresponding to Listing 3.1 is shown in Listing 3.2. The parameter `n` is passed in argument register `x10`. Since this is a non-leaf procedure, both the return address and argument must be preserved on the stack.

```
fact:
    addi sp, sp, -8     // adjust stack for 2 items
    sw   x1, 4(sp)       // save return address
    sw   x10, 0(sp)      // save argument n

    addi x5, x10, -1     // x5 = n - 1
    bge  x5, x0, L1      // if (n - 1) >= 0, go to L1

    addi x10, x0, 1      // return 1
    addi sp, sp, 8      // pop stack
    jalr x0, 0(x1)       // return

L1:
    addi x10, x10, -1    // argument = n - 1
    jal  x1, fact        // recursive call

    addi x6, x10, 0      // save result of fact(n-1)
    lw  x10, 0(sp)      // restore original n
    lw   x1, 4(sp)       // restore return address
    addi sp, sp, 8      // pop stack

    mul  x10, x10, x6    // n * fact(n-1)
    jalr x0, 0(x1)       // return
```

Listing 3.2: Recursive factorial in RISC-V assembly

Table 4.1 summarizes what is preserved across a procedure call. Note that several schemes preserve the stack, guaranteeing that the caller will get the same data back on a load from the stack as it stored onto the stack. The stack above sp is preserved simply by making sure the callee does not write above sp; sp is itself preserved by the callee adding exactly the same amount that was subtracted from it; and the other registers are preserved by saving them on the stack (if they are used) and restoring them from there.

Preserved	**Not preserved**
Saved registers: x8–x9, x18–x27	Temporary registers: x5–x7, x28–x31
Stack pointer register: x2 (sp)	Argument/result registers: x10–x17
Frame pointer: x8 (fp)	
Return address: x1 (ra)	
Stack above the stack pointer	Stack below the stack pointer

Table 4.1: Preserved and Non-Preserved Registers

Iterative Implementation

Some recursive procedures can be implemented iteratively without using recursion. Iteration can significantly improve performance by removing the overhead associated with recursive procedure calls. For example, consider a procedure used to accumulate a sum:

```c
long long int sum(long long int n, long long int acc) {
    if (n > 0)
        return sum(n - 1, acc + n);
    else
        return acc;
}
```

Listing 3.3: Tail-recursive summation in C

Consider the procedure call sum(3,0). This will result in recursive calls to sum(2,3), sum(1,5), and sum(0,6), and then the result 6 will be returned four times. This recursive call of sum is referred to as a tail call, and this example use of tail recursion can be implemented very efficiently (assume x10 = n, x11 = acc, and the result goes into x12):

```
sum:
    ble  x10, x0, sum_exit   // if n <= 0, exit
    add  x11, x11, x10       // acc = acc + n
    addi x10, x10, -1        // n = n - 1
    jal  x0, sum             // loop

sum_exit:
    addi x12, x11, 0         // return acc
    jalr x0, 0(x1)           // return to caller
```

Listing 3.4: Iterative summation in RISC-V

Task 1

Translate the C function shown in Listing 1 into RISC-V assembly to calculate the factorial using the iterative approach. How does the stack get populated? Observe and comment on the working of code.

Task 2

The nth triangular number is like a factorial but instead of a product we use a summation: `https://en.wikipedia.org/wiki/Triangular_number`

```c
int ntri(int num) {
    if (num <= 1)
        return 1;
    return num + ntri(num - 1);
}
```

Listing 3.5: Recursive computation of the Nth triangular number in C

Translate the above C function to calculate the Nth Triangular number into RISC-V assembly. You should also write the wrapper code, i.e., which calls this function with an argument and then displays the result returned by this function.
Since there's recursion going on, you will need to adjust the stack so that you store the return 4 address as well as any useful variables of the caller on the stack before calling the callee. Show the result computed. How does the stack get populated? Observe and comment on the working of code.

Task 3- Bubble Sort

```c
void bubble(int *a, unsigned int len) {

    if (a == NULL || len == 0)
        return;

    for (int i = 0; i < len; i++) {
        for (int j = i; j < len; j++) {
            if (a[i] < a[j]) {
                int temp = a[i];
                a[i] = a[j];
                a[j] = temp;
            }
        }
    }
    return;
}
```

Listing 3.6: Bubble sort implementation in C

Write the equivalent RISC-V assembly code for Listing 6. a and len get passed in x10 and x11 respectively.

Task 4 - Student-Selected Program

In this task, students will design a RISC-V assembly program of their own choice that demonstrates the concepts covered in this lab, such as nested procedures, stack management, and control flow.

Students are encouraged to select a program that can be extended or reused in their project. Proper use of procedure calls, register conventions, and stack discipline will be assessed. Students should clearly demonstrate correct execution of the program.

Assessment Rubric
Lab 4: Implementing Nested Procedures and Sorting

Name:	**Student ID:**	**Section:**

Points Distribution

	Task No.	LR 2 Code	LR 5 Results
In - Lab	**Task 1**	/10	/5
	Task 2	/10	/5
	Task 3	/15	/10
	Task 4	/15	/10
Total Points: 100		**/50**	**/30**
CLO Mapped		CLO 2	

Affective Domain Rubric		Points	CLO Mapped
AR7	**Report Submission / Git Upload**	/10 & /10	CLO 2

CLO	Total Points	Points Obtained
2	100	
Total	**100**	

For description of different levels of the mapped rubrics, please refer to the Lab Evaluation Assessment Rubrics and Affective Domain Assessment Rubrics provided here.

Lab 5: Pre lab - Synthesizable vs Non-Synthesizable Verilog

Pre-Lab Concept: Synthesizable vs Non-Synthesizable Verilog

Key Concept

Verilog can be used for **two very different purposes**:

- **Simulation**: To model and test behavior
- **Synthesis**: To generate real hardware on an FPGA

Not all Verilog constructs that work in simulation can be translated into physical hardware.

Key idea: *If you cannot imagine a piece of hardware implementing it, it is probably not synthesizable.*

1. Loops in Verilog: Simulation vs Synthesis

Example

Synthesizable `for` Loop

- Loop bounds are constant
- Loop represents replicated hardware

```
integer i;
always @(*) begin
  for (i = 0; i < 8; i = i + 1)
    y[i] = a[i] & b[i];
end
```

Hardware meaning: 8 parallel AND gates inferred by the synthesizer.

Common Pitfall

Non-Synthesizable Loop

```
always @(*) begin
  i = 0;
  while (a[i] != 1'b1)
    i = i + 1;
end
```

Why this fails synthesis:

- Unknown iteration count
- Hardware size cannot change dynamically

2. Timing Controls and Delays

Common Pitfall

Simulation-Only Delay

```
always begin
  clk = 0;
  #5 clk = 1;
  #5;
end
```

Arbitrary delays are not supported in FPGA hardware.

Example

Synthesizable Alternative

```
always @(posedge clk) begin
  q <= d;
end
```

Clock timing is provided by external oscillators or FPGA clocking resources.

3. Reset and Initialization

Common Pitfall

Simulation-Only Initialization

```
initial begin
  q = 0;
end
```

`initial` blocks are unreliable for sequential logic in synthesis.

Example

Synthesizable Reset

```
always @(posedge clk) begin
  if (reset)
    q <= 0;
  else
    q <= d;
end
```

Reset logic ensures a known hardware startup state.

4. Continuous Assignment vs Procedural Assignment

Example

Continuous Assignment

```
assign y = a & b;
```

Pure combinational logic.

Common Pitfall

Multiple Drivers (Illegal)

```
assign y = a & b;

always @(*) begin
  y = a | b;
end
```

Two hardware drivers for one signal cause synthesis failure.

Exercises (Pre-Lab)

Exercise

Exercise 1: Identify Synthesizability

```
always @(*) begin
  for (i = 0; i < n; i = i + 1)
    sum = sum + a[i];
end
```

Exercise

Exercise 2: Fix the Code

```
always begin
  #10 clk = ~clk;
end
```

Exercise

Exercise 3: What hardware does this code describe?

```
always @(posedge clk) begin
  if (en)
    q <= d;
```

```
end
```

From RTL to FPGA: Bitstream Generation

FPGA Design Flow

(a) **Synthesis**: RTL → FPGA primitives (LUTs, FFs)

(b) **Implementation (Place & Route)**

(c) **Timing Analysis**

(d) **Bitstream Generation**

(e) **FPGA Programming via USB/JTAG**

Key Concept

Simulation verifies functional correctness. Only synthesis and implementation guarantee reliable FPGA operation.

Lab 5: Designing FSM Using FPGA Switches and LEDs

Objective

The objective of this lab is to design, implement, and verify a Finite State Machine (FSM) on an FPGA using input switches, LEDs, and a reset button. Students will learn how to interface user inputs with hardware logic, implement state-based control, and manage synchronous counters using Verilog HDL.

By the end of this lab, students will be able to:

- Design an FSM using a state diagram
- Interface FPGA switches and LEDs with control logic
- Implement a decrement counter controlled by FSM states
- Integrate provided switch, button debouncing, and LED interface modules
- Verify FSM behavior through FPGA synthesis and on-board testing

System Description

The system behavior is defined as follows:

- The system continuously monitors the FPGA switches for input.
- If the switch input value is **zero**, the system remains in the input-waiting state.
- When the switch input becomes **non-zero**, the FSM:
  - Captures the switch value
  - Displays the value on the LEDs
  - Starts a decrement counter from the captured value down to zero
- While counting down:
  - The LEDs reflect the current counter value
  - Switch inputs are ignored
- When the counter reaches zero:
  - The FSM automatically returns to the input-waiting state
- At any time during counting, pressing the **reset button**:
  - Immediately returns the FSM to the input-waiting state
  - Clears the counter and LED output

Note: The switch interface, button debouncing interface, and LED interface are provided and must be used without modification.

Task 1: FSM Design and State Diagram

Objective

To design an FSM that controls system behavior based on switch input, counter value, and reset signal.

Procedure

(a) Identify the required FSM states, such as:
  - Input Waiting State
  - Countdown State
  - Reset State (if implemented explicitly)

(b) Draw a complete FSM state diagram showing:
  - State transitions
  - Conditions for transitions (switch input, counter reaching zero, reset)
  - Outputs associated with each state

(c) Clearly indicate:
  - How switch input is latched
  - When the counter is enabled or disabled
  - How LEDs are driven in each state

Deliverables

- FSM state diagram with labeled transitions and states

Task 2: Verilog FSM and Counter Implementation

Objective

To implement the FSM and decrement counter in Verilog HDL and verify functionality through simulation.

Procedure

(a) Implement the FSM in Verilog:
  - Use synchronous state transitions triggered by the system clock
  - Include an asynchronous or synchronous reset as specified

(b) Implement a decrement counter:
  - Load the counter with the switch input value
  - Decrement the counter on each clock cycle while enabled

  - Stop counting when the counter reaches zero

(c) Integrate the provided modules:

  - Switch interface:

```
module switches(
    input clk,
    input rst,
    input [31:0] writeData,
    input writeEnable,
    input readEnable,
    input [29:0] memAddress,

    output reg  [31:0] readData = 0, // not to be read
    output reg [15:0] leds
);
...
endmodule
```

  - Button debouncing interface:

```
module debouncer(
    input clk,
    input pbin,
    output pbout
```

  - LED interface:

```
module leds(
    input clk, rst,
    input [15:0] btns,
    input [31:0] writeData, //  not to be written
    input writeEnable, // not to be used
    input readEnable,
    input [29:0] memAddress,
    input [15:0] switches,

    output reg  [31:0] readData
```

(d) Develop a testbench to:

  - Apply different switch input values
  - Verify FSM transitions
  - Confirm correct countdown behavior
  - Verify reset functionality

Deliverables

- Verilog source file for FSM and counter
- Testbench file
- Simulation waveforms showing correct operation

Task 3: FPGA Constraints, XDC File and Timing Analysis

Objective

To introduce students to FPGA pin constraints, clock constraints, and timing analysis using Vivado synthesis reports.

Background

FPGA designs require a constraint file (XDC – Xilinx Design Constraints) to define:

- Which physical FPGA pins connect to design signals
- What clock frequency the design must meet
- Input/output timing requirements

Without a correct XDC file, synthesis cannot guarantee correct operation.

XDC Constraint File

Each top-level FPGA signal must be mapped to a physical FPGA pin.

Example Pin Constraint (Switch)

Map Switch Input to FPGA Pin

```
set_property PACKAGE_PIN V17 [get_ports {switches[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {switches[0]}]
```

Explanation:

- `PACKAGE_PIN V17` specifies the physical FPGA pin
- `IOSTANDARD LVCMOS33` defines voltage level (3.3V CMOS)

Example Pin Constraint (LED)

Map LED Output to FPGA Pin

```
set_property PACKAGE_PIN U16 [get_ports {leds[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {leds[0]}]
```

Clock Constraint

Define System Clock (100 MHz)

```
create_clock -period 10.000 -name sys_clk [get_ports clk]
```

Explanation: The clock period of 10 ns corresponds to 100 MHz. All logic must complete within this time.

Understanding Timing Slack

Slack = Required Time − Actual Path Delay

Slack Value	Meaning	Status
Positive	Timing met	PASS
Zero	Critical timing	WARNING
Negative	Timing violation	FAIL

Real Timing Report Examples

Example 1 – Good Design (PASS)

```
Slack (MET) : 3.215ns
Required Time : 10.000ns
Path Delay : 6.785ns
```

Interpretation: Design runs safely at 100 MHz.

Example 2 – Marginal Design (CRITICAL)

```
Slack (MET) : 0.042ns
Required Time : 10.000ns
Path Delay : 9.958ns
```

Interpretation: Design barely meets timing and may fail under temperature/voltage variation.

Example 3 – Failing Design (VIOLATION)

```
Slack (VIOLATED) : -1.372ns
Required Time : 10.000ns
Path Delay : 11.372ns
```

Interpretation: Design cannot operate at 100 MHz — must be optimized or clock slowed.

Procedure

(a) Add an XDC file to your project
(b) Assign pin constraints to all switches, LEDs, reset, and clock
(c) Add a clock constraint
(d) Run synthesis
(e) Report your Worst Negative Slack (WNS)

Deliverables

- XDC file
- Screenshot of Vivado Timing Summary
- WNS value and interpretation (PASS / MARGINAL / FAIL)

Task 4: FPGA Synthesis and On-Board Verification

Objective

To synthesize the FSM design on an FPGA and verify functionality using physical switches, LEDs, and reset button.

Procedure

(a) Integrate the FSM module into the top-level FPGA design.
(b) Assign FPGA pins for:
    - Switch inputs
    - LED outputs
    - Reset button
(c) Synthesize the design and analyze:
    - RTL schematic
    - State transitions
    - Counter implementation
(d) Program the FPGA and perform on-board testing:
    - Apply zero and non-zero switch values
    - Observe LED countdown behavior
    - Press reset during countdown to verify immediate reset
(e) Demonstrate correct operation to the lab instructor or RA.

Deliverables

- RTL schematic screenshots
- Successful FPGA demonstration

Expected Learning Outcomes

Upon successful completion of this lab, students will be able to:

- Design FSMs for hardware control applications
- Interface user inputs with synchronous digital systems
- Implement counters controlled by FSM logic
- Debug and verify FPGA-based designs using real hardware

Assessment Rubric
Lab 5: Designing FSM Using FPGA Switches and LEDs

Name:	**Student ID:**	**Section:**

Points Distribution

	Task No.	LR 2 (Code)	LR 5 (Results)	LR 11 (Design)
In-Lab	Pre lab	-	/15	-
	Task 1	-	-	/10
	Task 2	/20	/20	-
	Task 3	-	/15	-
Total Points		**/20**	**/50**	**/10**
CLO Mapped		CLO 1		

Affective Domain Rubric		Points	CLO Mapped
AR7	**Report Submission & Git Upload**	/10 & /10	CLO 1

CLO	Total Points	Points Obtained
1	100	
Total	**100**	

For description of different levels of the mapped rubrics, please refer to the Lab Evaluation Assessment Rubrics and Affective Domain Assessment Rubrics provided here.

Lab 6: Design and FPGA Implementation of the ALU for a Single-Cycle RISC-V Processor

Objective

To design and implement the ALU for a single-cycle RISC-V processor (RV32I base extension) using Verilog HDL, verify its functionality through a simulation testbench, synthesize the design on an FPGA, and demonstrate its operation using an FSM-based verification approach. The ALU should support the following core RISC-V operations:

ADD, SUB, SLL, SRL, AND, OR, XOR, BEQ

Task 1: ALU Block Diagram Design

Objective

To create a block diagram of the ALU showing the inputs, outputs, and internal operations.

Procedure

(a) Design the 1-bit ALU block diagram showing:
- Arithmetic operations: ADD, SUB
- Logic operations: AND, OR, XOR
- Shift operations: SLL, SRL
- Input multiplexers (if needed for operand selection)

(b) Design the 32-bit ALU block diagram showing:
- Arithmetic operations: ADD, SUB
- Logic operations: AND, OR, XOR
- Shift operations: SLL, SRL

(c) Draw clear diagrams showing data flow from inputs through the operation blocks to outputs.

Deliverables

- Block diagram of the ALU.

Task 2: Verilog Description and Testbench Development

Objective

Implement the ALU in Verilog based on the block diagram and verify its functionality using simulation.

Procedure

(a) Implement the ALU module in Verilog:
- Inputs: `A`, `B`, `ALUControl[3:0]`
- Outputs: `ALUResult`, `Zero`

(b) Follow the block diagram to implement arithmetic, logic, and shift operations.

(c) Develop a testbench:
- Apply test vectors for all operations (ADD, SUB, AND, OR, XOR, SLL, SRL, etc.).
- Verify outputs match the expected results.
- Generate waveforms for visual verification.

(d) Run simulation to ensure correctness of all ALU operations.

Deliverables

- Verilog source file for ALU (`ALU.v`)
- Testbench file (`ALU_tb.v`)
- Simulation waveforms showing correct operation

Task 3: FPGA Synthesis and RTL Analysis

Objective

Synthesize the ALU design on an FPGA, analyze the RTL structure, and verify functionality using switches and LEDs.

Procedure

(a) Integrate ALU module into a top-level design.

(b) Implement a simple FSM to cycle through ALU operations.

(c) Map ALUControl input to FPGA switches.

(d) Map ALU outputs (`ALUResult`, `Zero`) to LEDs.

(e) Use two fixed 32-bit numbers (e.g., `0x10101010`, `0x01010101`) as operands A and B.

(f) Use the FSM to read switch inputs and display ALU results on LEDs.

(g) Observe LED patterns to confirm correct ALU operations.

(h) Synthesize the design and verify the RTL schematic.

(i) Program the FPGA, toggle switches, and verify LEDs reflect expected outputs.

Deliverables

- RTL schematic screenshots
- Demonstration of FPGA operation to RA for approval

Note: The switch interface, button debouncing module, and LED interface must be re used from Lab 5.

Assessment Rubric
Lab 6: Design and FPGA Implementation of the ALU for a Single-Cycle RISC-V Processor

Name:	**Student ID:**	**Section:**

Points Distribution

	Task No.	LR 2 (Code)	LR 5 (Results)	LR 11 (Design)
In-Lab	Task 1	-	-	/20
	Task 2	/20	/15	-
	Task 3	/10	/15	-
Total Points		**/30**	**/30**	**/20**
CLO Mapped		CLO 1		

Affective Domain Rubric		Points	CLO Mapped
AR7	**Report Submission & Git Upload**	/10 & /10	CLO 1

CLO	Total Points	Points Obtained
1	100	
Total	**100**	

For description of different levels of the mapped rubrics, please refer to the Lab Evaluation Assessment Rubrics and Affective Domain Assessment Rubrics provided here.

Lab 7: Design and FPGA Implementation of the Register File for a Single-Cycle RISC-V Processor

Objective

To design and implement the 32×32 register file (RV32I convention) using Verilog HDL, verify its functionality through a simulation testbench, synthesize the design on an FPGA, and demonstrate its operation by building an FSM that exercises both the ALU and the register file together. The register file should follow RISC-V conventions (register x0 hardwired to 0, two asynchronous read ports, one synchronous write port) and the FSM should perform a sequence of reads/writes and ALU operations to validate correct integration.

Task 1: Register File Block Diagram Design

Objective

To create a block diagram of the register file showing inputs, outputs, internal features (x0 behavior), and how it connects to the ALU.

Procedure

(a) Design the full 32×32 register file block diagram showing:

- Inputs: `clk`, `rst`, `WriteEnable`, `rs1[4:0]`, `rs2[4:0]`, `rd[4:0]`, `WriteData[31:0]`
- Outputs: `ReadData1[31:0]`, `readData2[31:0]`
- Internal x0 handling: address 0 always returns 0 and writes to address 0 are ignored.

(b) Draw clear diagrams showing data flow:

- Show how `writeData` is routed to the selected register on a rising clock edge when `WriteEnable` is asserted.
- Show how `rs1` and `rs2` select registers for combinational outputs `readData1` and `reaData2`.
- Show connection points to the ALU: `readData1` and `readData2` feed the ALU's `A` and `B` inputs (or one of them).

Deliverables

- Block diagram of the 32×32 register file and its integration points with the ALU.

Task 2: Verilog Description and Testbench Development

Objective

Implement the register file in Verilog based on the block diagram and verify its functionality using simulation. Then develop an integrated testbench that uses an FSM to test the ALU and register file together.

Procedure

(a) Implement the Register File module in Verilog:

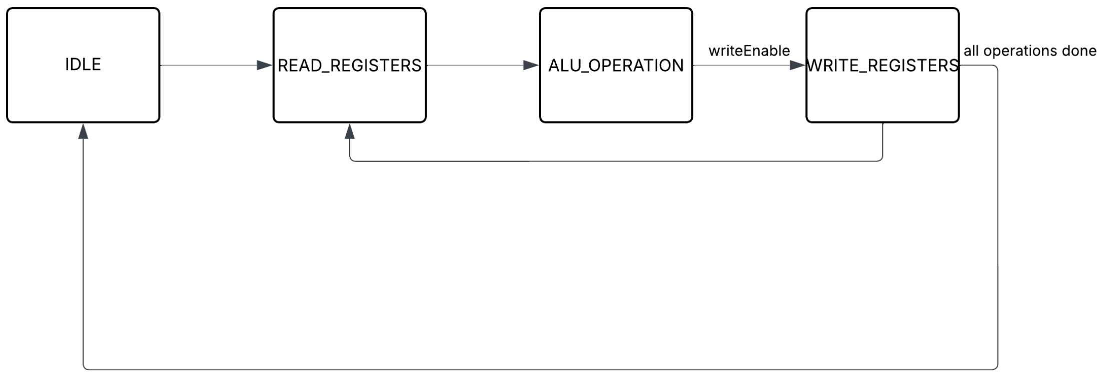
- Enforce RISC-V semantics: register x0 (address 0) always reads as 32'b0 and writes to x0 are ignored.
- Use an array of registers (`reg [31:0] regs [31:0]` or similar)

(b) Develop a testbench for the register file only (`RegisterFile_tb.v`):

- Test cases:
  i. Write a value to a register (e.g., x5 = 0xDEADBEEF) with `WriteEnable=1` and check on next clock that `readData1` or `readData2` returns the same when addressed.
  ii. Attempt to write to x0 and verify it remains zero.
  iii. Simultaneous two read ports: verify both reads return the correct contents concurrently.
  iv. Overwrite a register and verify old value is replaced.
  v. Reset behavior: if a synchronous/asynchronous reset is implemented, check registers clear appropriately.
- Generate waveforms.

(c) Implement an integrated testbench that exercises the Register File and ALU together with an FSM (`RF_ALU_FSM_tb.v`):

- Instantiate the previously implemented ALU module (from the ALU lab) and the Register File.
- Build an FSM that cycles through a deterministic sequence of states to:
  i. Write known constants into several registers (e.g., x1 = 0x10101010, x2 = 0x01010101, x3 = 0x00000005).
  ii. Read x1 and x2, feed them to the ALU, perform ADD, SUB, AND, OR, XOR, SLL, SRL operations, and write ALU results into destination registers (e.g., x4..x10).
  iii. Perform a BEQ-style check using ALU Zero output and conditionally write a flag register.
  iv. Test register read-after-write timing (write a register and then read it in the following cycle).
- The FSM should include clearly named states (e.g., `IDLE`, `WRITE_REGS`).



- Produce waveforms capturing the register file contents, ALU inputs/outputs, and FSM state transitions.

(d) Run simulation to ensure correctness of all behaviors:

- Register read/write correctness and x0 behavior.
- ALU results when fed from register file outputs.
- FSM transitions and conditional actions based on ALU Zero flag.

Deliverables

- Verilog source file for Register File (`RegisterFile.v`)
- Testbench file for Register File (`RegisterFile_tb.v`)
- Integrated FSM + ALU + Register File testbench (`RF_ALU_FSM_tb.v`)
- Simulation waveforms showing correct operation and assertions printed for passed tests

Task 3: FPGA Synthesis and RTL Analysis

Objective

Synthesize the Register File and the FSM+ALU integration on an FPGA, analyze the RTL structure, and verify functionality using switches, LEDs, and (optionally) a small seven-segment display for numeric output.

Procedure

(a) Integrate Register File and ALU into a top-level design (`top_rf_alu.v` or similar):

- Instantiate the Register File and the ALU.
- Add an FSM module that repeats the verification sequence used in simulation.
- Map switches to force `WriteEnable`, `rd`

(b) Map inputs and outputs to FPGA I/O:
  - Map a small set of switches to choose which test or which ALU operation to run, and to set control values including `writeEnable`.
  - Map LEDs to show FSM state.

(c) Use two fixed 32-bit constants for quick verification (example):
  - Constant A = `32’h10101010`
  - Constant B = `32’h01010101`
  - Use FSM to write these into registers (e.g., x1 and x2) at startup.

(d) Synthesize the design and analyze the RTL schematic:
  - Inspect the register array implementation, write-enable logic, and decoder.

(e) Program the FPGA and verify on-board:
  - Press the configured switches to choose an ALU operation and observe LED patterns showing the result (lower bits) and the FSM state.
  - Run the FSM demo sequence and demonstrate to RA.

Deliverables

- RTL schematic screenshots (register file array, decoder, FSM, ALU connections)
- Demonstration of FPGA operation to RA for approval
- Top-level Verilog file (`top_rf_alu.v`).

Note: The switch interface, button debouncing module, and LED interface must be re used from Lab 5.

Assessment Rubric
Lab 7: Design and FPGA Implementation of the Register File for a Single-Cycle RISC-V Processor

Name:	**Student ID:**	**Section:**

Points Distribution

	Task No.	LR 2 (Code)	LR 5 (Results)	LR 11 (Design)
In-Lab	Task 1	-	-	/10
	Task 2	/20	/10	-
	Task 3	/20	/20	-
Total Points		**/40**	**/30**	**/10**
CLO Mapped		CLO 1		

Affective Domain Rubric		Points	CLO Mapped
AR7	**Report Submission & Git Upload**	/10 & /10	CLO 1

CLO	Total Points	Points Obtained
1	100	
Total	**100**	

For description of different levels of the mapped rubrics, please refer to the Lab Evaluation Assessment Rubrics and Affective Domain Assessment Rubrics provided here.

Lab 8: Design and FPGA Implementation of Memory System with Address Decoding for a Single-Cycle RISC-V Processor

Objective

To design and implement the data memory and address decoding for a single-cycle RISC-V processor using Verilog HDL, verify functionality through simulation, synthesize the design on an FPGA, and demonstrate operation using switches and LEDs.

Proposed Memory Map

- Data Memory: 0–511
- LEDs: 512–767
- Switches: 768–1023

Task 1: Design of Data Memory and Address Decoder

Objective

To design a data memory module and an address decoder that routes CPU memory accesses to the correct memory or peripheral.

Address Decoding

Address decoding is the mechanism by which the processor determines which memory or peripheral device must respond to a given memory access. Since the CPU presents a single unified address bus, address decoding ensures that only one device is activated at a time based on the address value.

In this memory system, three address-mapped devices are present:

- Data Memory
- LED Output Interface
- Switch Input Interface

Although the processor generates a 32-bit (or 64-bit) memory address, only the 10 least significant bits of the address are used in this lab to form a local memory map. This allows addressing up to $2^{10} = 1024$ memory-mapped locations.

The address format is shown below:

```
address[9:0] = [Device Select (2 bits) | Local Address (8 bits)]
```

- `address[9:8]` selects which device is accessed.
- `address[7:0]` selects the specific register or memory location inside that device.

Device Selection

address[9:8]	Address Range	Device	Enable Signal
00	0–511	Data Memory	DataMemSelect
01	512–767	LEDs	LEDSelect
10	768–1023	Switches	SwitchSelect
11	–	Unused	None

Decoder Operation

The address decoder continuously monitors the CPU address and generates internal enable signals based on `address[9:8]`:

- When accessing Data Memory, the decoder enables memory read/write.
- When accessing LEDs, the decoder enables LED write only.
- When accessing Switches, the decoder enables switch read only.

Only one device is enabled at any time, preventing bus contention and ensuring correct routing of data.

Example

If the CPU issues the address:

$$\texttt{address = 0x210 = } 528_{10}$$

$$\texttt{address[9:8] = 01}$$

The decoder enables the LED module, and the lower 8 bits (`address[7:0]`) determine which LED register is accessed.

Similarly, for:

$$\texttt{address = 0x350 = } 848_{10}$$

$$\texttt{address[9:8] = 10}$$

The Switch module is selected.

This decoding scheme provides a simple, scalable, and hardware-efficient way to implement memory-mapped I/O in the single-cycle RISC-V processor.

Procedure

(a) Design Data Memory Module:

  - Inputs: `clk`, `MemWrite`, `address`, `write_data`
  - Output: `read_data`
  - Implement as a 512 × 32 memory array.

(b) Use LED and Switch modules as provided in Lab 5.

(c) Design Address Decoder using `address[9:8]` to generate:

  - DataMemWrite, DataMemRead
  - LEDWrite
  - SwitchReadEnable

(d) Draw a block diagram showing CPU, Address Decoder, Data Memory, LEDs, and Switches.

Deliverables

- Block diagram of memory system with address decoding.

Task 2: Verilog Implementation and Testbench Development

Objective

To implement the data memory and address decoder in Verilog and verify functionality through simulation.

Procedure

(a) Implement Data Memory and Address Decoder modules in Verilog. The top module, in direct communication with the processor, may appear as follows:

```
module addressDecoderTop(
    input clk, rst,
    input [31:0] address,
    input readEnable, writeEnable,
    input [31:0] writeData,
    input [15:0] switches,

    output [31:0] readData,
    output [15:0] leds
);
...
endmodule
```

(b) Integrate LED and Switch modules provided in Lab 5.

(c) Develop a testbench:

- Write to Data Memory and LEDs.
- Read from Data Memory and Switches.
- Verify that the address decoder enables only the correct module for each access.

(d) Run simulation, generate waveforms, and verify:

- Data Memory stores and outputs correct data
- LEDs reflect written values
- Switch values are read correctly

Deliverables

- Verilog source files: `DataMemory.v`, `AddressDecoder.v`
- Testbench file: `MemorySystem_tb.v`
- Simulation waveforms showing correct memory and peripheral access.

Task 3: FPGA Synthesis and RTL Analysis

Objective

To synthesize the memory system on the FPGA, analyze the RTL structure, and verify operation using provided switches and LEDs.

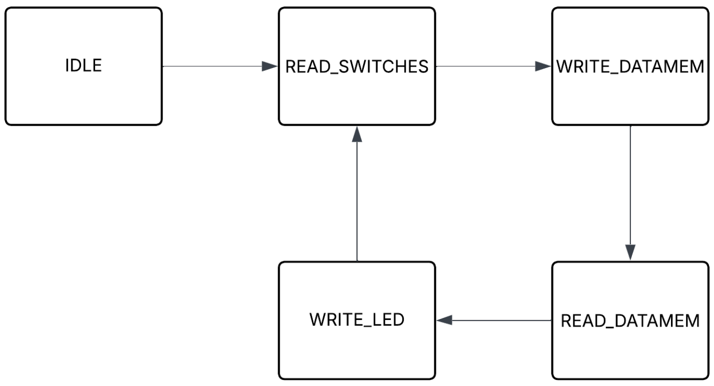
Procedure

(a) Integrate Data Memory, LEDs, Switches, and Address Decoder into a top-level design.

(b) Map memory and peripheral connections to FPGA pins.

(c) Test memory operations using switches or a simple FSM:

- Write to Data Memory and LEDs
- Read from Data Memory and Switches



(d) Observe LEDs to verify correct writes; confirm switch values are read properly.

(e) Synthesize the design and inspect the RTL schematic.

(f) Program the FPGA and verify memory-mapped operations interactively.

Deliverables

- RTL schematic screenshots
- Demonstration of FPGA setup showing correct memory and peripheral operation

Note: The switch interface, button debouncing module, and LED interface must be re used from Lab 5.

Assessment Rubric
Lab 8: Design and FPGA Implementation of Memory System with Address Decoding for a Single-Cycle RISC-V Processor

Name:	**Student ID:**	**Section:**

Points Distribution

	Task No.	**LR 2 (Code)**	**LR 5 (Results)**	**LR 11 (Design)**
In-Lab	Task 1	-	-	/20
	Task 2	/20	/15	-
	Task 3	/10	/15	-
Total Points		**/30**	**/30**	**/20**
CLO Mapped		CLO 1		

Affective Domain Rubric		**Points**	**CLO Mapped**
AR7	**Report Submission & Git Upload**	/10 & /10	CLO 1

CLO	**Total Points**	**Points Obtained**
1	100	
Total	**100**	

For description of different levels of the mapped rubrics, please refer to the Lab Evaluation Assessment Rubrics and Affective Domain Assessment Rubrics provided here.

Lab 9: Design and FPGA Implementation of the Control Path for a Single-Cycle RISC-V Processor

Objective

The goal of this laboratory exercise is to design and implement the control path of a single-cycle RISC-V (RV32I) processor using Verilog HDL. The design will be verified through simulation and synthesized on an FPGA for hardware demonstration. The control unit must support the following core RISC-V instructions:

ADD, ADDI, SUB, SLL, SRL, AND, OR, XOR, LW, LH, LB, SW, SH, SB, BEQ

Additionally, students will verify the implemented design using an FSM-based hardware verification approach.

Task 1: Developing the Control Path Truth Table

Objective

To determine the relationship between RISC-V instruction fields and the control signals required in a single-cycle datapath, expressed in the form of a truth table.

Procedure

1. Review the RISC-V Instruction Set Format

- Study the R-type, I-type, S-type, and B-type instruction formats.
- Identify the relevant fields: `opcode`, `funct3`, and `funct7`.

2. List Supported Instructions

The design must support the following RV32I instructions:

ADD, ADDI, SUB, SLL, SRL, AND, OR, XOR, LW, LH, LB, SW, SH, SB, BEQ

3. Identify Required Control Signals

Determine the control outputs needed for the datapath:

RegWrite, ALUSrc, MemRead, MemWrite, MemtoReg, Branch, ALUOp

4. Construct the Truth Tables

For each instruction, determine the correct setting of control signals based on the required datapath behavior. Use logic values:

$$1,\ 0,\ X \text{ (don't care)}$$

Deliverables

- A truth table for the main control unit (opcode as input, control signals as outputs).
- A truth table for the ALU control unit (ALUOp, funct3, funct7 as inputs).

Task 2: Verilog Description and Testbench Development

Objective

To implement the main control unit and ALU control logic in Verilog and verify their functionality through simulation using a testbench. The implementation must follow the truth tables created in Task 1.

Procedure

1. Implement the Main Control Unit

- **Inputs:** `opcode[6:0]`
- **Outputs:** RegWrite, ALUOp, MemRead, MemWrite, ALUSrc, MemtoReg, Branch

2. Implement the ALU Control Unit

- **Inputs:** ALUOp[1:0], `funct3`, `funct7`
- **Output:** ALUControl

3. Handle Don't-Care Conditions

Ensure that "don't care" (X) values from the truth table are assigned default or safe values in Verilog.

4. Develop the Testbench

The testbench must verify *every* supported instruction and generate waveforms for inspection.

5. Run Simulation

Simulate the design, generate waveforms, and verify correctness of the control signal transitions.

Deliverables

(a) Verilog source files for the Main Control and ALU Control modules.

(b) Testbench Verilog files.

(c) Simulation waveform screenshots highlighting all control signal transitions.

Task 3: FPGA Synthesis and RTL Analysis

Objective

To synthesize the RISC-V control path on an FPGA, examine the resulting RTL schematic, and verify functionality using physical switches and LEDs.

Procedure

(a) Integrate the MainControl and ALUControl modules into a top-level design.

(b) Implement a simple finite-state machine (FSM) inside the top-level module for verification.

(c) Connect input fields (`opcode`, `funct3`, `funct7`) to FPGA switches.

(d) Connect output control signals (RegWrite, MemRead, MemWrite, ALUSrc, MemtoReg, Branch, ALUControl, etc.) to LEDs.

(e) Use the FSM to read switch positions and update LED displays accordingly.

(f) Observe LED patterns to confirm that correct control signals are generated for each instruction.

(g) Synthesize the design and inspect the RTL schematic.

(h) Program the FPGA and verify correct operation by toggling switch inputs.

Deliverables

- RTL schematic screenshots.
- Demonstration of FPGA setup to the RA for approval.

Note: The switch interface, button debouncing module, and LED interface must be re used from Lab 5.

Assessment Rubric
Lab 9: Design and FPGA Implementation of the Control Path for a Single-Cycle RISC-V Processor

Name:	**Student ID:**	**Section:**

Points Distribution

	Task No.	**LR 2 (Code)**	**LR 5 (Results)**	**LR 10 (Analysis)**
In-Lab	Task 1	-	-	/20
	Task 2	/20	/15	-
	Task 3	/10	/15	-
Total Points		**/30**	**/30**	**/20**
CLO Mapped		CLO 1		

Affective Domain Rubric		**Points**	**CLO Mapped**
AR7	**Report Submission & Git Upload**	/10 & /10	CLO 1

CLO	**Total Points**	**Points Obtained**
1	100	
Total	**100**	

For description of different levels of the mapped rubrics, please refer to the Lab Evaluation Assessment Rubrics and Affective Domain Assessment Rubrics provided here.

Lab 10: State-Based Control Flow Using RISC-V Assembly

Objective

The objective of this lab is to implement a Finite State Machine (FSM) entirely in assembly language using switches and LEDs as memory-mapped I/O devices. Students will apply previously learned concepts of address decoding while focusing on control flow design, stack usage, and instruction execution.

This lab reinforces software–hardware interaction by requiring students to:

- Implement FSM behavior using assembly-level control flow
- Use conditional branching to represent FSM state transitions
- Employ the stack to support function calls and register preservation
- Store and execute assembly instructions from an instruction memory module

The functional behavior of the FSM remains identical to that implemented in Lab 5.

Processor Clock Frequency and Real-Time Behavior

Processor Clock Rate

In this lab, the RISC-V processor operates at a fixed clock frequency of:

$$f_{clk} = 10\ \text{MHz}$$

This means the processor executes:

$$10,000,000 \text{ clock cycles per second}$$

Every instruction, memory access, branch, and loop iteration consumes one or more clock cycles. Therefore, software execution time directly depends on the clock frequency.

Why This Matters

This directly affects:

- Delay loops
- Software timers
- Counter-based delays
- Busy-wait routines

For example, a loop that runs 100 million iterations will take:

$$\frac{100,000,000}{10,000,000} = 10 \text{ seconds}$$

This means poorly designed delay loops can easily cause unexpected long delays or register overflows.

Design Implication for This Lab

Since the FSM countdown and polling loops are implemented in assembly, students must account for clock-dependent execution time to ensure responsive and correct behavior.

System Description

The system behavior is defined as follows:

- Switches, LEDs, and the reset button are accessed as memory-mapped locations.
- The system continuously reads the switch input value.
- If the switch input value is **zero**, the system remains in the input-waiting state.
- When the switch input value becomes **non-zero**:
  - The value is captured
  - The value is displayed on the LEDs
  - A decrement counter begins, counting from the captured value down to zero
- During the countdown:
  - The LEDs display the current counter value
  - Further switch input is ignored
- When the counter reaches zero:
  - The FSM automatically returns to the input-waiting state
- Pressing the reset button at any time during the countdown:
  - Immediately resets the FSM to the input-waiting state
  - Clears the counter and LED output

Note: Students are expected to use memory-mapped I/O knowledge acquired in the address decoding lab.

Task 1: FSM Design in Assembly

Objective

To translate the FSM logic into an assembly-based control flow using labels, branches, and state variables.

Procedure

(a) Review the FSM designed in Lab 5.

(b) Identify the required states, such as:

- Input Waiting State

- Countdown State

(c) Represent FSM states using:
- State variables stored in registers or memory
- Assembly labels and conditional branches

(d) Draw a control flow diagram showing:
- State transitions
- Branch conditions
- Reset behavior

Deliverables

- FSM control flow diagram for the assembly implementation

Task 2: Assembly Implementation of the FSM

Objective

To implement the complete FSM behavior in assembly language using control flow, stack-based function calls, and memory-mapped I/O.

Procedure

(a) Write an assembly program that:
- Continuously reads switch input values
- Implements FSM state transitions using conditional branches
- Displays values on LEDs according to the current FSM state

(b) Implement the countdown logic as a subroutine:
- Pass the initial count value as an argument
- Decrement the value until zero
- Update LED output at each step

(c) Use the stack to:
- Save and restore registers
- Preserve return addresses during subroutine calls

(d) Ensure reset detection immediately returns execution to the input-waiting state.

Deliverables

- Complete assembly source file implementing the FSM
- Demonstration of correct stack usage in the code

Task 3: Instruction Memory Module Design

Objective

To design and implement an instruction memory module that stores and supplies the FSM assembly instructions to the processor.

Procedure

(a) Design an instruction memory module in Verilog:

- Store the assembled machine code instructions
- Output instructions based on the program counter (PC)

(b) Initialize the instruction memory using the reference provided:

```
module instructionMemory#(
    parameter OPERAND_LENGTH = 31
    )(
    input [OPERAND_LENGTH:0] instAddress,
    output reg [31:0] instruction
);
reg [7:0] memory [0:255];
...
endmodule
```

(c) Populate the instruction memory with the machine language translation of the assembly code of Task 2.

(d) Verify correct instruction sequencing during execution.

Deliverables

- Verilog instruction memory module
- Memory initialization file containing assembly instructions

Assessment Rubric
Lab 10: State-Based Control Flow Using RISC-V Assembly

Name:	**Student ID:**	**Section:**

Points Distribution

	Task No.	LR 2 (Code)	LR 5 (Results)	LR 11 (Design)
In-Lab	Task 1	-	-	/20
	Task 2	/20	/20	-
	Task 3	/20	-	-
Total Points		**/40**	**/20**	**/20**
CLO Mapped		CLO 1		

Affective Domain Rubric		Points	CLO Mapped
AR7	**Report Submission & Git Upload**	/10 & /10	CLO 1

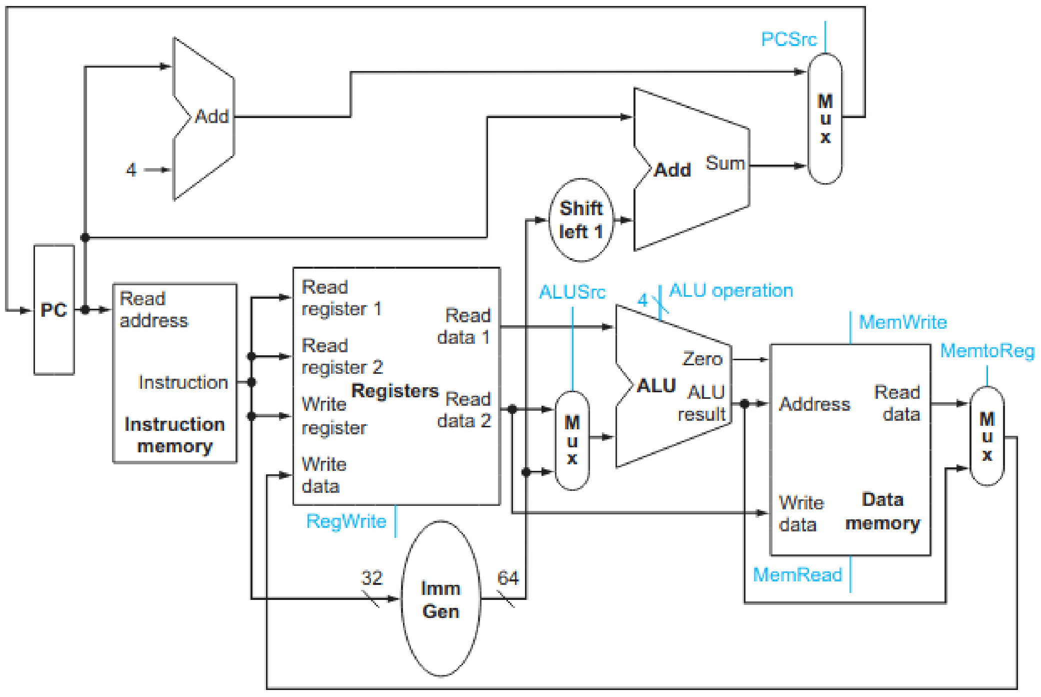
CLO	Total Points	Points Obtained
1	100	
Total	**100**	

For description of different levels of the mapped rubrics, please refer to the Lab Evaluation Assessment Rubrics and Affective Domain Assessment Rubrics provided here.

Lab 11: Design and FPGA Implementation of a Single-Cycle RISC-V Processor

Objective

To integrate all previously designed processor components into a complete single-cycle RISC-V (RV32I) processor. Students will design the Instruction Memory, connect all datapath modules, verify the processor through simulation, and deploy it on an FPGA. The final processor will execute a given RISC-V program stored in instruction memory, producing observable results via LEDs or seven-segment displays.



Task 1: Program Counter, Immediate Generator and Branch Infrastructure

Objective

To design and verify the Program Counter update mechanism and immediate generation logic required for sequential execution and branch control in the RISC-V processor.

System Overview

In a single-cycle RISC-V processor, the Program Counter (PC) must update on every clock cycle to either:

- The next sequential instruction address (PC + 4), or
- A branch target address (PC + imm $\ll$ 1) when a branch is taken.

To support this functionality, the processor requires the following infrastructure modules.

Required Modules

Module	Function
ProgramCounter	Stores and updates the PC value
pcAdder	Computes PC + 4
branchAdder	Computes PC + (imm $\ll$ 1)
mux2	Selects next PC value
immGen	Generates I, S and B type immediates

Module Requirements

- PC must update on every positive clock edge.
- pcAdder must add a constant value of 4 to the PC.
- branchAdder must compute PC + (sign-extended immediate $\ll$ 1).
- mux2 must select between sequential and branch target addresses using PCSrc.
- immGen must support I, S and B type immediate formats.

Procedure

(a) Create the modules: `ProgramCounter.v`, `pcAdder.v`, `branchAdder.v`, `mux2.v`, and `immGen.v`.

(b) Implement immediate generation according to the RV32I encoding format.

(c) Connect all modules to form the PC update datapath.

(d) Write a testbench to verify:

   - PC increments correctly by 4 when PCSrc = 0.
   - PC updates to branch target when PCSrc = 1.
   - Immediate generation produces correct signed values.

Deliverables

- Verilog files: `ProgramCounter.v`, `pcAdder.v`, `branchAdder.v`, `mux2.v`, `immGen.v`
- Testbench waveform showing correct PC update and immediate generation

Task 2: Datapath Integration and Top-Level Processor

Objective

To integrate the instruction memory with the remaining datapath components to form the complete single-cycle RISC-V processor. Students will connect PC, control logic, register file, ALU, immediate generator, data memory, and program flow components into one unified design.

Procedure

(a) Instantiate all modules inside a new file named `TopLevelProcessor.v`.

(b) Connect instruction memory output to the control unit and register file inputs.

(c) Connect control signals to ALU, ALU control, register file, and data memory.

(d) Verify all datapath signals match the RISC-V single-cycle architecture.

Deliverables

- Verilog file: `TopLevelProcessor.v`
- Complete datapath diagram schematic
- Utilization report
- Timing Analysis report

Assessment Rubric
Lab 11: Design and FPGA Implementation of a Single-Cycle RISC-V Processor

Name:	**Student ID:**	**Section:**

Points Distribution

	Task No.	OE2 (Design)	OE5 (Report)	OE6 (Reflection)
In-Lab	Task 1	/40	-	
	Task 2	-	/30	-
	Viva	-	-	/10
Total Points		**/40**	**/30**	**/10**
CLO Mapped		CLO 3		

Affective Domain Rubric		Points	CLO Mapped
AR7	**Report Submission & Git Upload**	/10 & /10	CLO 3

CLO	Total Points	Points Obtained
3	100	
Total	**100**	

For description of different levels of the mapped rubrics, please refer to the Open Ended Lab rubrics provided here.

Course Project: Assembly-Level Execution on Integrated RISC-V Processor

Objective

The objective of this project is to integrate RISC-V assembly language programming with a fully functional single-cycle RISC-V processor implemented on hardware. Students will execute the assembly program developed in Lab 10 on the integrated processor designed in Lab 11, demonstrating a clear understanding of instruction execution, datapath behavior, and hardware–software interaction.

Project Requirements

Part A: Base Assembly Program

Students must use the RISC-V assembly program developed in Lab 10 as the starting point. The program must be correctly loaded into the instruction memory of the integrated processor from Lab 11 and executed on hardware. Correct execution must be demonstrated through visible hardware outputs.

Part B: Instruction Extension

Students are required to implement **any three (3)** RISC-V instructions that are **not already included** and are of **different types** in the provided processor code. Each selected instruction must be properly verified using assembly-level testing. All added instructions must be demonstrated on hardware.

Part C: Program Demonstration on Hardware

In addition to the base program and instruction extensions, students must demonstrate at least one meaningful assembly-level program executing on the processor.

Examples include, but are not limited to:

- Fibonacci series computation
- Factorial calculation
- Summation of an array
- Loop-based arithmetic sequence

The program output must be observable on hardware, and students must be able to explain the execution flow and results.

Implementation Guidelines

- All modifications must be integrated into the existing single-cycle RISC-V processor.

- Instruction memory must be updated to include the extended assembly program.
- The design must synthesize without errors or critical warnings.
- Hardware execution must correctly reflect instruction sequencing and expected results.

Deliverables

- Updated RISC-V assembly program
- Modified Verilog source files (control logic and datapath, if required)
- Instruction memory initialization
- Simulation waveforms demonstrating correct execution
- FPGA hardware demonstration
- Brief explanation of implemented instructions and program behavior

RV32I Base Integer Instruction Set

The following table lists the supported instructions from the **RV32I** base integer instruction set. Students may use these instructions when implementing and extending the processor for labs and the course project.

Note: This table includes only the **RV32I base integer instruction set**. No extension instructions (such as `M`, `A`, `F`, or `D`) are included or required for this course.

Instruction	Description	Format
Load Instructions		
LB	Load Byte (signed)	I-type
LH	Load Halfword (signed)	I-type
LW	Load Word	I-type
LBU	Load Byte Unsigned	I-type
LHU	Load Halfword Unsigned	I-type
Store Instructions		
SB	Store Byte	S-type
SH	Store Halfword	S-type
SW	Store Word	S-type
Immediate Arithmetic Instructions		
ADDI	Add Immediate	I-type
SLTI	Set Less Than Immediate (signed)	I-type
SLTIU	Set Less Than Immediate (unsigned)	I-type
XORI	XOR Immediate	I-type
ORI	OR Immediate	I-type
ANDI	AND Immediate	I-type
SLLI	Shift Left Logical Immediate	I-type
SRLI	Shift Right Logical Immediate	I-type
SRAI	Shift Right Arithmetic Immediate	I-type
Register-Register Arithmetic Instructions		
ADD	Add	R-type
SUB	Subtract	R-type
SLL	Shift Left Logical	R-type
SLT	Set Less Than (signed)	R-type
SLTU	Set Less Than (unsigned)	R-type
XOR	XOR	R-type
SRL	Shift Right Logical	R-type
SRA	Shift Right Arithmetic	R-type
OR	OR	R-type
AND	AND	R-type
Branch Instructions		
BEQ	Branch if Equal	B-type
BNE	Branch if Not Equal	B-type
BLT	Branch if Less Than (signed)	B-type
BGE	Branch if Greater or Equal (signed)	B-type
BLTU	Branch if Less Than Unsigned	B-type
BGEU	Branch if Greater or Equal Unsigned	B-type
Jump Instructions		
JAL	Jump and Link	J-type
JALR	Jump and Link Register	I-type
Upper Immediate Instructions		
LUI	Load Upper Immediate	U-type
AUIPC	Add Upper Immediate to PC	U-type
System and Fence Instructions		
ECALL	Environment Call	–
EBREAK	Environment Break	–
FENCE	Memory Fence	–
FENCE.I	Instruction Fence	–

Project Evaluation Rubric

The course project will be evaluated based on the following criteria. Emphasis is placed on correct hardware execution, instruction implementation, creative program demonstration, and individual accountability through version control.

Criterion	**Description**	**Weight**	**EA**
Hardware Execution: Lab 10 Assembly Program	Correct execution of the RISC-V assembly language program developed in Lab 5 on the integrated single-cycle RISC-V processor. Instructions must execute in the correct sequence and produce observable results on hardware (e.g., LEDs or seven-segment displays). Students must clearly demonstrate and explain hardware behavior.	**20%**	EA2
Creativity and Program Demonstration	**Design** and **verification** of a meaningful RISC-V assembly language program beyond the base requirements. Evaluation focuses on program structure, correct use of instructions, stack usage, register usage, and logical correctness. Hardware execution is not assessed under this criterion.	**10%**	EA3
Hardware Execution: Student - Selected Program	Successful execution of a student-selected RISC-V assembly program on the FPGA-based processor, with mandatory and correct stack usage for procedure calls and register preservation. The program must execute correctly on hardware and produce verifiable output, demonstrating correct datapath and control operation.	**20%**	EA2
Implementation of Additional Instruction 1	Correct implementation of one new RISC-V instruction not included in the provided processor code. The instruction must be properly decoded, executed through the datapath, and verified. All changes made must be clearly documented and explained.	**5%**	EA3
Implementation of Additional Instruction 2	Correct implementation of a second new RISC-V instruction, including necessary control logic and datapath modifications. Functionality must be validated through simulation and hardware demonstration.All changes made must be clearly documented and explained.	**5%**	EA3
Implementation of Additional Instruction 3	Correct implementation of a third new RISC-V instruction, demonstrating understanding of instruction format, control signals, and execution behavior. The instruction must execute correctly on hardware.All changes made must be clearly documented and explained.	**5%**	EA3
Individual Contribution	Viva-based assessment of individual understanding, focusing on RISC-V processor architecture and hardware synthesis concepts (Chapter 1). Students must demonstrate clear knowledge of tasks, design choices, and implemented components during oral evaluation.	**20%**	–
Git Repository Management	Proper use of version control through a shared Git repository, including regular and meaningful commits, clear commit messages, and timely uploads of source code and documentation.	**10%**	EA2
Report Submission	Proper use of version control through a shared Git repository, including regular and meaningful commits, clear commit messages, and timely uploads of source code and documentation.	**10%**	EA2
CLO Mapped	CLO-3		

Complex Engineering Activity

This project is a CEA and is mapped to the following Washington Accord (WA) attributes

Code	**Activity Characteristic**
EA1	Range of resources – use of diverse, possibly specialized, resources
EA2	Level of interactions – requires managing significant interactions with stakeholders or components
EA3	Level of innovation – includes adaptation or development of technologies
EA4	Consequences – outcomes have significant consequences in economic, health, safety, or environmental terms
EA5	Familiarity – activities involve unpredictability and require judgment and responsibility