# Documentation

# CSCI\_6461\_12: Project P1

## Group 10:

### Alekya Kowta

### Nikhil Arethiya

### Peiyan Liu

### Aryan Saxena

# Overview and Purpose

This project phase establishes the functional core of the C6461 CPU Simulator and its associated assembler. The primary purpose was to achieve complete compliance with the C6461 Instruction Set Architecture (ISA) for all non-floating point operations.

The simulator now accurately models the architecture by implementing all core arithmetic, logical, load/store, and instructions specified in ISA. The system is fully capable of loading and executing any valid user program designed for the C6461 core instruction set.

# GitHub Repository Link

<https://github.com/AlekyaKowta/CSCI_6461_FALL25_PROJECT-TEAM-10>

# Installation and Usage

Installation:

* Ensure the target system has Java Runtime Environment (JRE) version 8 or higher installed.
* Download or obtain the packaged Simulator.jar file.
* Ensure the input file (LoadFile) you want to provide is in the same directory as the jar file.

Usage:

* Open a terminal or command prompt.
* Navigate to the directory containing Simulator.jar and LoadFile.txt.
* Run the assembler with the command:

A black text on a white background

AI-generated content may be incorrect.

 **Load Program:** Once the GUI appears, click the **IPL** button. This will open a file dialog, prompting you to select your program's **Load File**.

 **Execute:** After loading, *the PC is set to the program's starting address*. Use the **Step** button for instruction-by-instruction debugging or **Run** to execute the program continuously.

# Folders

* Documentation folder: Holds documents
* UI Folder: Has the SimulatorUI code
* Core: Has Machine Controller and MachineState
* Assembler: Has previous Assembler and Opcode

# User Interface and Register Manipulation

A screenshot of a computer

AI-generated content may be incorrect.

The simulator's user interface acts as the **front panel** for the machine, allowing the operator to read the current state and manually deposit data into registers and memory locations.

### I. Reading and Checking Register Status

All register fields are updated automatically by the MachineControlelr after every step, run, load, or store operation. To check the current status:

* **PC and MAR:** Read the 4-digit octal value to determine the next instruction address PC or the current memory access address MAR
* **GPRs and IXRs:** Read the 6-digit octal value to check the contents of the general-purpose data registers or the base addresses in the index registers.
* **MFR and CC:** These fields display the 4-bit status of the MFR and CC registers.

### II. Inputting Data into Registers (Depositing)

To manually change the value of any primary register (R0-R3, X1-X3, PC, MAR, MBR), you must use the central OCTAL INPUT field in combination with the corresponding register's **small, square Load button**.

|  |  |  |
| --- | --- | --- |
| Step | Action | Resulting Flow |
| 1. Set Value | Enter the desired octal value (up to 6 digits) into the central OCTAL INPUT field. | This value is now prepared for deposit. |
| 2. Select Destination | Locate the specific register field you want to modify (e.g., PC or GPR1). |  |
| 3. Deposit Value | Click the small, square Load button positioned directly next to the target register field. | The value from the OCTAL INPUT is deposited into that register. Note: Values deposited into PC or MAR are automatically masked to 12 bits. |
| 4. Verify | The register's display field updates instantly to reflect the new, deposited value. | The Console Log records the action (Manual Load: PC set to ...). |

# Input Specification

The simulator requires a **Load File** as input, which is the direct numerical output from the two-pass assembler. This file defines the memory addresses and their contents.

|  |  |  |
| --- | --- | --- |
| Component | Format | Description |
| Input File | LoadFile.txt | A plain text file where each non-empty line contains exactly two values separated by whitespace (spaces or tabs). |
| Address Field | 4-6 octal digits | The memory address where the value should be placed. Addresses must be valid (0 to 2047 installed words). |
| Value Field | 6 octal digits | The 16-bit word (data or instruction) to be deposited at the specified address. |
| Example Line | 000016 102207 | Stores the instruction 1022078​ into memory address 0000168​. |

# Output Specification

The simulator primarily generates two types of output for debugging and status tracking: Console Log (Printer Area) and Machine State updates.

|  |  |  |
| --- | --- | --- |
| Content | Format | Purpose |
| IPL Status | Text | Confirms successful loading of the Load File and indicates the initial PC setting. |
| Execution Trace | PC=0016: Op=41, ... | Logs every executed instruction, showing the PC location, decoded fields, and the resulting action. |
| Fault Log | FAULT: Illegal Opcode | Displays error messages when a Machine Fault is triggered. |
| Console Actions | Console Load/Store | Records manual register and memory inspection actions performed by the operator. |

All register displays are strictly enforced based on the C6461 ISA:

|  |  |  |
| --- | --- | --- |
| Register Type | Format | Display Size |
| PC, MAR (12-bit registers) | Octal (%04o) | 4 octal digits (0000 to 7777). |
| GPRs, IXRs, MBR, IR (16-bit registers) | Octal (%06o) | 6 octal digits (000000 to 177777). |
| MFR, CC (4-bit registers) | Octal (%04o) | 4 octal digits for consistent UI display (only 2 are significant). |

# Design and Structure

### Key Architectural Components

|  |  |  |
| --- | --- | --- |
| Component | Class/Structure | Purpose |
| Instruction Handlers | Modular Functions | Modular functions (handleArithmeticLogic, handleShiftRotate, lsInstructionParse) that manage the complex bit-shifting and operand encoding for specific instruction formats. |
| Simulator Core | MachineState | Models all hardware registers and memory, providing the essential state management for the MachineController. |
| Execution Flow | MachineController | Orchestrates the Fetch-Decode-Execute cycle and enforces all Machine Faults and 12-bit address masking required by the ISA. |

### Execution Principles

The simulator's structure ensures strict compliance with the C6461 hardware specifications:

* **Modular Hardware Access:** The MachineController never accesses registers directly; it uses the MachineState methods (PC,GPR to ensure all state changes are tracked and logged.
* **Enforced Compliance:** Crucial architectural constraints, such as the 12-bit limit on the PC and MAR, are enforced by masking in the MachineController to prevent illegal memory addressing.
* **Full ISA Coverage:** Modular instruction handlers ensure that all core instructions—from simple LDR complex MLT and RRC are implemented with their correct bit positions and execution logic.

### Instruction Implementation

The design uses a switch statement in singleStep to route execution to dedicated, modular instruction handlers (e.g., handleLDR, handleJMA).This structure confirms that **all core instructions** are implemented with their correct bit positions, register pairing (like MLT) and condition code update logic required by the ISA.

## Execution and Memory Maintenance Controls

These buttons are located in the main control panel and are essential for executing programs and manually inspecting or modifying memory.

Execution Control

|  |  |  |
| --- | --- | --- |
| Button | Action | Purpose in Simulation |
| Run | Executes the instruction cycle continuously in a loop until a HLT instruction is encountered or a Machine Fault is detected (MFR=0). | Used for running the full program or code segments without manual stepping. |
| Step | Executes exactly one complete Fetch-Decode-Execute cycle. | Ideal for debugging and observing register state changes after each instruction. |
| Halt | Stops any ongoing Run operation immediately. | Used to interrupt a continuous program execution for inspection. |

Memory Inspection and Deposit

|  |  |  |  |
| --- | --- | --- | --- |
| Button | Action | Register Flow | Purpose |
| Load | Reads the content of memory location M[MAR] and copies it into the MBR. | M[MAR]→MBR | Inspects the word at the current MAR address. |
| Load+ | Performs the Load operation, and then automatically increments the MAR (MAR←MAR+1). | M[MAR]→MBR;MAR++ | Quickly steps through and inspects consecutive memory locations. |
| Store | Writes the content of the MBR into the memory location M[MAR]. | MBR→M[MAR] | Deposits data or an instruction word into memory. |
| Store+ | Performs the Store operation, and then automatically increments the MAR (MAR←MAR+1). | MBR→M[MAR];MAR++ | Quickly deposits data into a block of consecutive memory locations. |

# Error Handling

The calculateEA method serves as a critical **Security Gate** before any memory access occurs during execution.

* **Memory Faults:** This function checks the calculated Effective Address (EA against three MFR (Machine Fault Register) conditions defined in the ISA: IllegalOpcode, ReservedMemoryAccess (MFR=0001), and Memory Beyond Bounds.
* **Immediate Halt:** Upon detection of any fault, the machine immediately halts and sets the appropriate MFR code for debugging.

# Extensibility

* New instructions added by updating switch statements and adding handler functions.
* Modularity facilitates future enhancements without major rewrites.

# Limitations

* **Advanced Status Handling:** While the arithmetic instructions (AMR, MLT etc.) perform the calculation, the logic for precisely detecting and setting the **Condition Codes** for **Overflow**, **Underflow**, and **Division by Zero** is simplified or incomplete (to be implemented in P2).
* **Interrupts and Traps:** The TRAP instruction is not implemented, and the system does not handle external Interrupts.
* **I/O Implementation:** The IN, OUT, CHK instructions are currently handled by placeholder functions that interact with the console output area but do not simulate specific device behaviors or detailed status checks.

# Testing

|  |  |  |  |
| --- | --- | --- | --- |
| Test Case | Description | Expected Outcome | Verification |
| Full Sample Program | Loading the LoadFile and executing sequentially from the start of the instruction segment (PC=0016) to the final HLT instruction (PC=2000). | Program should successfully perform LDR with Indexing/Indirection, set R0=0 via LDA, and correctly execute the conditional jump (JZ) to the HLT label. | Confirmed final log output matches the expected sequence of EA calculations and register state changes, terminating at PC=2000. |
| IPL Start Verification | Starting execution at M[0006], which contains the data word 0000128​. | The machine must execute M[0006] as an instruction, decode it as OpCode 0 (HLT), and immediately halt. | Confirmed IPL Warning is logged, and machine halts on PC=0006. |

|  |  |  |
| --- | --- | --- |
| Test Focus | Method Used | Expected Result |
| 12-bit PC/MAR | Manual Load/Store+ operations using addresses over 409510​. | MAR must be masked to 12 bits (MAR←MAR AND 0xFFF), preventing addressing beyond the 4096 word limit. |
| Reserved Memory Access | Attempting LDR with EA=00028​ during singleStep. | Machine halts, and MFR is set to 00012​ (Illegal Memory Reserved fault). |
| Bounds Violation | Executing an instruction that calculates EA≥2048 (the simulator's installed memory size). | Machine halts, and MFR is set to 10002​ (Memory Beyond Bounds fault). |

# References

* Instruction Set Architecture Specification Document
* Java Development Kit Documentation
* Related Academic Publications on Assembly Language Programming