IP Tile – Jesús Esparza

## Overview

This tile was designed by the student Jesús Esparza as part of a custom microcontroller (uC) implementation. The design represents a minimalist yet functional 8-bit microcontroller, tailored for didactic and experimental purposes. It features a reduced instruction set composed of 16 essential operations, enabling memory access, I/O handling, control flow, and arithmetic/logic execution.

The internal architecture is structured around a simplified two-state finite state machine (FSM), which transitions between:

* FETCH – retrieves the instruction from program memory and decodes.
* EXECUTE – performs the operation.

**Instruction Set**

The instruction format uses a 16-bit word, segmented into an opcode and three additional fields (registers or immediate values, depending on the operation). The complete instruction set is summarized below:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| INSTRUCTION | DATA[15:0] | | | | DESCRIPTION |
| OPCODE | BYTE 2 | BYTE 1 | BYTE 0 |
| NOP | 0x0 | x | x | x | Does nothing. Typically used for delay or alignment. |
| LOAD | 0x1 | reg\_dst | ram\_addr\_src | | Loads a value from memory (ram\_addr\_src) into reg\_dst |
| STORE | 0x2 | reg\_src | ram\_addr\_dst | | Stores the contents of reg\_src into memory at ram\_addr\_dst. |
| JMP | 0x3 | pc\_dst | | | Unconditionally jumps to program counter address pc\_dst |
| BEQ | 0x4 | pc\_dst | | | Jumps to pc\_dst if the last comparison (CMP) was equal. |
| BC | 0x5 | pc\_dst | | | Jumps to pc\_dst if a carry-out occurred in the last add (ADD). |
| IN | 0x6 | reg\_dst | data\_in\* | | In bootstrap mode data\_in storages in reg\_dst, if it is not bootstrap mode reads data from input port and stores it in reg\_dst, and data\_in don’t care (xx). |
| OUT | 0x7 | reg\_src | x | out\_port | Sends the contents of reg\_src to an output port (PORT0 or PORT1). |
| ADD | 0x8 | reg\_dst | reg\_a | reg\_b | Adds reg\_a and reg\_b; stores the result in reg\_dst. Sets the carry flag. |
| SUB | 0x9 | reg\_dst | reg\_a | reg\_b | Subtracts reg\_b from reg\_a; result is stored in reg\_dst. |
| AND | 0xA | reg\_dst | reg\_a | reg\_b | Performs bitwise AND on reg\_a and reg\_b; stores the result in reg\_dst. |
| OR | 0xB | reg\_dst | reg\_a | reg\_b | Performs bitwise OR on reg\_a and reg\_b; stores the result in reg\_dst. |
| NOT | 0xC | reg\_dst | reg\_a | x | Bitwise NOT on reg\_a; result is stored in reg\_dst. |
| CMP | 0xD | x | reg\_a | reg\_b | Compares reg\_a and reg\_b; updates equality flag but stores no result. |
| SHR | 0xE | reg\_dst | reg\_a | reg\_b | Right shifts reg\_a by reg\_b bits; result stored in reg\_dst. |
| SHL | 0xF | reg\_dst | reg\_a | reg\_b | Left shifts reg\_a by reg\_b bits; result stored in reg\_dst. |

**Memory Mapping**

The microcontroller implements a memory map to support dual-phase operation:

* Bootstrapping mode (addresses 0x000 to 0x1FF): This region is writable using the IN instruction and can be used to pre-load register values or instructions during the initialization phase.
* Execution mode (addresses 0x200 and above):  
  This region is dedicated to program execution. The control unit fetches and decodes instructions from this area once the system exits bootstrap mode.

**Microarchitecture**

The microcontroller consists of the following main components:

* Program Counter (PC): Generates instruction addresses, supports jumps and branches.
* ALU (Arithmetic Logic Unit): Supports 8-bit operations including addition, subtraction, logic gates, shifts, and comparisons.
* SRAM (Registers Bank): Allows read/write access to 256x8-bit memory block using dedicated control signals (write\_en, saddr, data\_in/out).
* GPIO: Facilitates communication with the outside world via in\_gpio and out\_gpio.
* Control Unit (CU): A finite-state machine that orchestrates instruction sequencing and data routing.

A diagram of a computer

AI-generated content may be incorrect.

## Usage Guide

**Folder Structure Requirement**

Make sure you execute the script from inside the 'equipo\_16' directory, where the following files are present:

* csr.py
* tests.py
* interface.py
* programs.py

**Usage**

Example command to run the default multiplication program:

*python tests.py --test mul --cycles 1000 --inputs 9 5*

**Command-Line Arguments**

|  |  |  |  |
| --- | --- | --- | --- |
| Argument | Type | Default | Description |
| --test | str | "mul" | Name of the test program to execute. Must match a keyword defined in programs.py. |
| --cycles | int | 1000 | Number of cycles the test will run. Controls duration. |
| --verbose | flag | False | Enables detailed output for debugging. |
| --inputs | int list | [9, 4] | List of integer inputs for the test program. |

**Important Note**

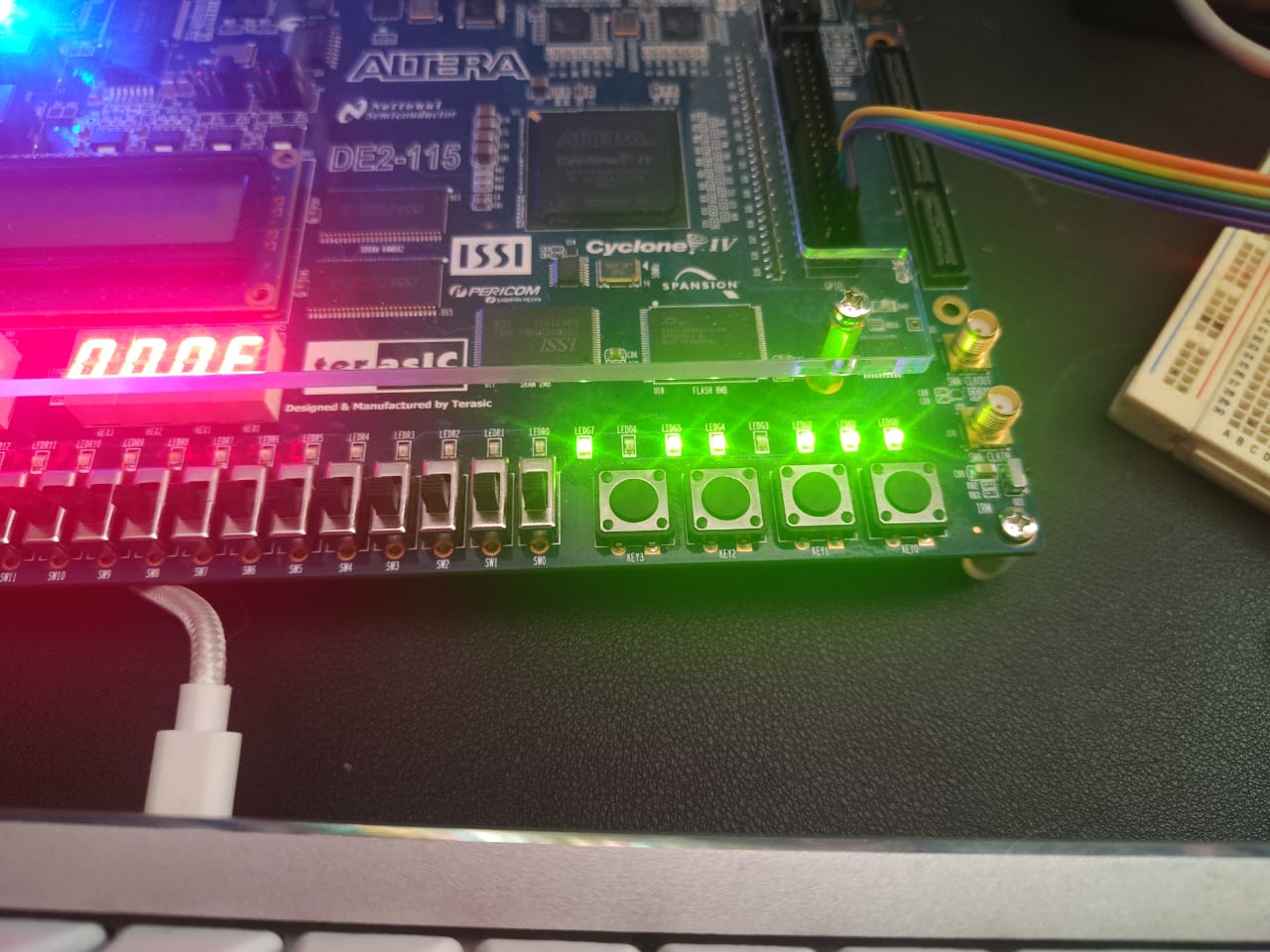
Before running any test, press the lower-right push button on the FPGA board to reset the system. This ensures proper synchronization over the UART interface.

**Case 1**

A computer screen with white text

AI-generated content may be incorrect.

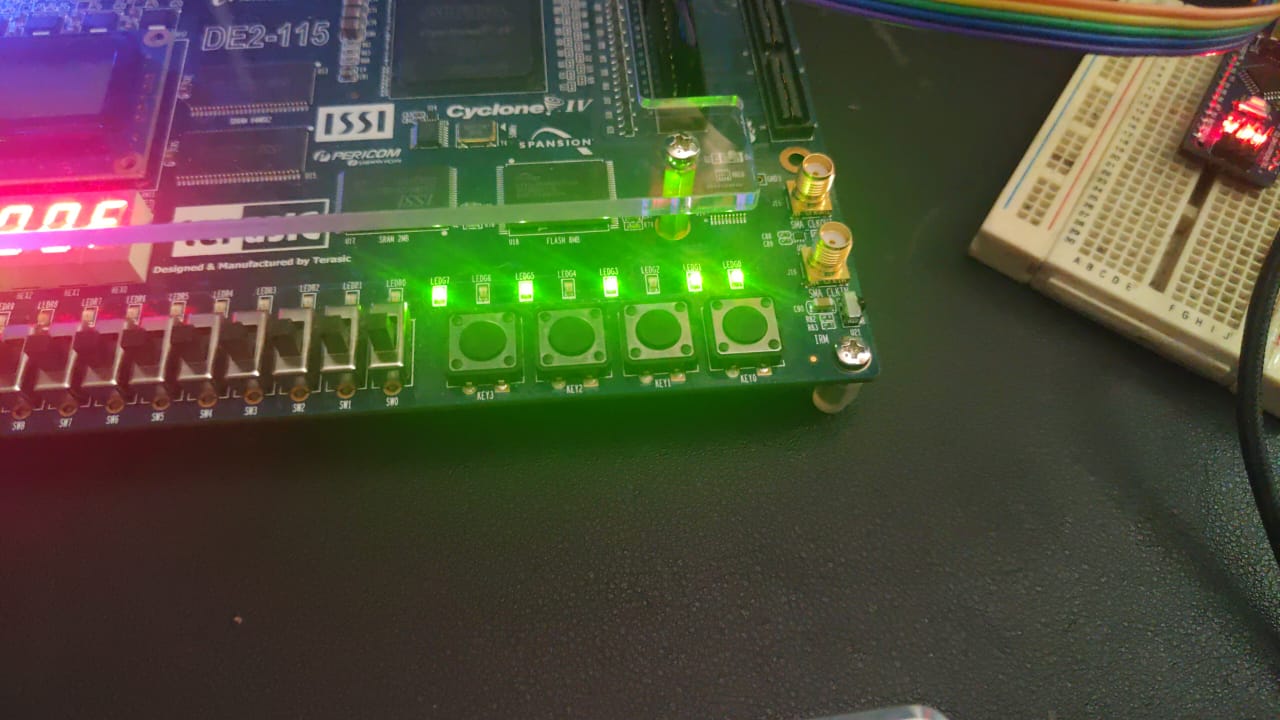
Inputs: 9, 5 → Result: 9 x 5 = 45 → Binary: 0b101101



**Case 2**A computer screen with white text

AI-generated content may be incorrect.

Inputs: 7, 6 → Result: 7 x 6 = 42 → Binary: 0b101010



**Custom Programs**

By default, only the 'mul' program is defined in programs.py. To create and run additional tests:

* Add a new keyword in programs.py and define the instruction sequence.
* Run: python tests.py --test <your\_keyword> --cycles 1000

**Purpose of the "mul" Program**

The 'mul' test program demonstrates how multiplication can be implemented using only 9 out of the 16 available instructions in the ISA: LOAD, STORE, IN, OUT, ADD, SUB, JMP, BEQ, and NOP. It showcases the uC’s capability to execute non-trivial logic using minimal instruction resources. The next picture show the program.

A screen shot of a computer program

AI-generated content may be incorrect.