User Manual for ItsyBitsy CPU
Presented to you by RacingBits Corporation

Pledge: I pledge my honor that I have abided by the Stevens Honor System.

1. Name of the CPU

ITSY-BITSY
This CPU is implemented entirely in Logisim-Evolution v4.0.0 and is designed as a small data path consisting of a 4-general purpose registers, an ALU that performs addition and multiplication, and RAM memory.

2. Owners and Designers

- **Aaditya Kulkarni**
- **Samdarshi Kumar Rai**

3. Job Descriptions

Job Description	Whodunnit
Circuit	**Aaditya Kulkarni**: Designed the data path architecture, constructed the register-file, implemented the ALU subsystem, routed multiplexer selections, and connected arithmetic components (Adder, Multiplier, XOR/AND/OR/NOT gates) and aligned the data path circuitry to the memory and decoder circuits. **Samdarshi Kumar Rai:** Implemented memory components (RAM and related control logic), decoder circuits for register selection, write-enable management circuitry, and performed debugging and verification of data path functionality. **Joint Contribution:** Deciding the opcodes for the CPU and how will script read the program and agreed on a common schematic to decode bits.
Assembler Script	Aaditya Kulkarni Reviewed by: Samdarshi Kumar Rai
Documentation	Samdarshi Kumar Rai Reviewed by: Aaditya Kulkarni (Yeah you guessed it right… we are understaffed)

4. How to Use the CPU

This CPU is intended to be simulated using Logisim-Evolution (preferably version 4.0 or later).

4.1 Opening the Project

1. Install Logisim-Evolution 4.0.0 or later and open it.
2. Select File (at the top left corner of the screen) → Open and locate the folder in the pop-up

window where the .circ file is saved.

4.2 Inputs and Outputs

- The CPU uses 8-bit binary encoding (specifications and details of which are mentioned ahead in this document) and hence 8-bit pins for data inputs and outputs.
- LED components indicate various internal signals (and a special message for the users as holidays are around).
- The register file contains wires to and from:
  - ReadReg1, ReadReg2
  - RegData1, RegData2
  - WriteReg, RegDataW
  - WriteEnable
- For our *precious* users, the CPU expects the user to assemble their program using the assembler provided by the RacingBits Corporation. The assembler will generate a txt file (or as known as imagefile in the world of LogiSim) and upload the output file generated by the assembler into the instruction fetching part of the CPU.
- How to upload the output file (aka imagefile): Find the label Instruction Fetching in the CPU (tentatively at the bottom left of the CPU) and right-click on the RAM in this section and select load image, then locate the folder in which the newly generated imagefile is located and select it to fill the RAM with the assembled instructions of your program. Now, the CPU is ready to roll out your program! **NOTE:** The users are advised the following:
  - For convenient and practical purposes, we advise users to store the .circ file, their program file and the assembler in the same folder. Failure to comply with aforesaid will result in assembling error as the assembler works only for the programs in the folder it is stored in.
  - We also strongly advise using the IDLE environment for Python to open our assembler as it provides hassle-free file I/O. Users might face some issues or in most of the cases a FileNotFoundError in VSCode if the VSCode software's files and libraries are not in the same folder as your program and the assembler.

4.3 Executing....Program!

1. Our assembler creates two output files, namely: text_output.hex and data_output.hex. It is according to the directives .text(this section has all the instructions written by you) and .data(this section has all the values you put in) respectively.
2. For .data segment: It is important that you put .text and .data in your program, if you have no data to put just put the following code so assembler know you have no data to input:

   .data

   0

   It is important to note that **the data in the .data segment does not relate to anything specific in the circuit or the program** unless and **until the user designs the program in a way that the data in .data segment becomes useful.**

3. For the .text segment: write all of your instructions in here and load the text_output file you get from the assembler into the RAM located in Instruction Fetching section.
4. Once you get both the output files and the assembled imagefiles are mounted in the instruction fetching RAM and memory access and writeback RAM, input the values into each of our powerful registers manually (select the finger pointer as your cursor in the Logisim environment and click on each register involved in the first line of instruction and input the desired data).
5. The clock component is located in the upper-left corner of the circuit. The user may:
    - Step it manually using the toggle switch under the "Simulate" tab on the top left corner of the simulator or using the keyboard shortcuts as shown in the corresponding options under "Simulate" tab, OR
    - Enable continuous simulation via Simulate → Auto-Tick Enabled.
    - Toggle the clock to start executing your program.

5. Architecture Description

5.1 Register File

The CPU contains four general-purpose 8-bit registers: X_0, X_1, X_2, X_3 (nobody wanted the exes… therefore, we took the opportunity!)

Each register supports synchronous writing using the global clock and a WriteEnable signal.

5.2 ALU

The ALU take the corresponding data from the registers used in the instruction and selects the right data using a multiplexor. The ALU supports the following operations:

- 8-bit addition (Adder)
- 8-bit multiplication (Multiplier)

A multiplexer selects the final ALU output based on the instruction in your assembled program.

5.3 Memory

Two RAM modules are included (one in the instruction fetching stage and one in the memory access stage):

- 8-bit data width
- Line-based enable configuration

Memory addressing is provided directly by register outputs.

5.4 Datapath Summary

The CPU is single-cycle and non-pipelined working off a global single clock, consisting of:

- Register file (4 registers)
- ALU

- Multiplexers for register and operation selection
- Two RAM blocks

6. Instruction Set and Formats

In this section, we define the instruction sets available and their respective syntax in our state-of-the-art language HOHOHO.
Our language in its current version does not support operations or instructions that involve immediate; therefore we are limited to instructions including registers.

6.1 Supported Operations and their CPU interpretations

Syntax interpretation by the CPU - OPCODE R_d, R_n, R_m

Binary bit representation - [7:6] [5:4] [3:2] [1:0]

Disclaimer and Warning: All the below operations you shall see below are programmed with the assumptions that the **values in the register are always positive integers.**

Instruction	Operation	Effective operation
PLUS	Addition	$R_d = R_n + R_m$
CROSS	Multiplication	$R_d = R_n * R_m$
LOAD	Loading data from memory	$R_d = M[R_n, R_m]$
STORE	Stores data from the register into memory	$M[R_n, R_m] = R_d$

Where $M[R_n, R_m]$ implies the contents at that memory address

6.2 Instructions

- Users need not worry about the formatting of the instruction here as well as similar to commas or the spacing between commas or the different registers as long as there are required number of registers included in the instruction (as all the instruction set is based on 3 registers, the below example is valid for all of them).
- For example, all of the following syntax are valid formats for the PLUS instruction:
  PLUS X_0, X_1, X_2
  PLUS X_0 X_1 X_2
  PLUS X_0 X_1 X_2
  PLUS X_0, X_1 X_2
  And any such variation that do not include any other character apart from the shown above is valid and will be compiled.
- Users can use "#" and "//" before writing out their comments in the program.

1. **PLUS R_d, R_n, R_m**

- This instruction adds the values in the registers R_n and R_m and stores the result in the destination register R_d. Given that all the values in all the registers are non-negative.

2. **CROSS R_d, R_n, R_m**
   - This instruction multiplies the values in the registers R_n and R_m and stores the result in the destination register R_d. Given that all the values in all the registers are non-negative.

3. **LOAD R_d, R_n, R_m**
   - This instruction loads the data from the memory into the register R_d with the address in the register R_n as base address and R_m as byte offset. Given that all the values in all the registers are non-negative.

4. **STORE R_d, R_n, R_m**
   - This instruction stores the data from the register R_d in the memory using the address in the register R_n as base address and R_m as byte offset. Given that all the values in all the registers are non-negative.

7. Instruction Encoding

Although the CPU does not use a physical instruction memory, the logical encoding is:

Syntax interpretation by the CPU - OPCODE R_d, R_n, R_m

Binary bit representation - [7:6] [5:4] [3:2] [1:0]

Opcode Mapping	Register Mapping
PLUS -> 01	X_0 -> 00
CROSS -> 10	X_1 -> 01
LOAD -> 11	X_2 -> 10
STORE -> 00	X_3 -> 11

8. Example Program

We have attached a sample program that compiles successfully for users to take a look at how to use the instructions. (users can choose whichever values they like to load into the registers)

```
.text
# Includes all 4 instructions, all registers, and spacing variations

PLUS X2, X1, X2       # X2 = X1 + X2
CROSS X1, X0, X3      # X1 = X0 * X3
LOAD X2, X3, X1       # X2 = M[X3, X1]
CROSS X0, X2, X1      # X0 = X2 * X1
PLUS X3 X2 X1         # X3 = X2 + X1
STORE X3 X1 X0        # M[X1, X0] = X3
PLUS   X2 ,   X2 X0   # X2 = X2 + X0
```

```
CROSS X0 ,X1,  X3      # X0 = X1 * X3
STORE X0, X2 , X0      # M[X2, X0] = X0

.data
0
1
2
34
5
6
```

9. Notes About the CPU Logisim File

The provided *.circ* loads all required Logisim libraries, sets toolbar presets, and places all components at fixed coordinates. No edits to the *.circ* are required for use; simulation is performed entirely by loading the image files created by the provided assembler and manual input of data in the registers for computation at each respective instruction.