Silverback CPU
User Manual

Emiliano Gutierrez

1. Name of the CPU

For this project, I designed and built a small single–cycle CPU that I ended up calling the **Silverback CPU**. The name doesn't have any deep meaning behind it—I just wanted something simple and cool, lol.

2. Partner

I didn't choose to have a partner due to the grades of the first project.

3. Job Description

Since I worked alone, all parts of the project were my responsibility. This included:

- designing the overall datapath,
- choosing the instruction format and encodings,
- wiring everything in Logisim-Evolution,
- writing the Python assembler,
- testing instructions and writing the demo program,
- putting together this user manual.

4. How to Use the Assembler

4.1 Running the Assembler

The assembler file is called `Silverback.py`. It reads a text file with my custom assembly language and outputs a hex file that Logisim understands.

Run it like this:

```
python3 Silverback.py demo.s demo.hex
```

The script doesn't use any special libraries and doesn't check for syntax errors, so the assembly program needs to be written correctly.

4.2 Loading the Machine Code Into Logisim

Once the hex file is created, you can load it:

1. Open `Silverback.circ`.
2. Click on the ROM component.
3. Choose "Load Image".
4. Select the hex file (e.g., `demo.hex`).

If your program uses loads, make sure RAM has the right initial values. You can click on RAM, go to "Edit Contents," and type in values before running the CPU.

5. CPU Architecture Description

5.1 Register File

Silverback has **four general-purpose registers**: R0, R1, R2, and R3. They each store an 8-bit value. The assembly language refers to them directly as `R0`–`R3`.

The datapath can read two registers in a single cycle and write to one, which is enough for the ADD and SUB formats I chose.

5.2 ALU

The ALU is simple but does what it needs to:

- **ADD**: Rn + Rm
- **SUB**: Rn - Rm

A single control bit decides which operation happens. I chose ADD and SUB because they're straightforward to wire and easy to test, but also because the project required at least two arithmetic operations.

5.3 Memory

There are two memory blocks:

- **ROM** (256x8) for instructions.
- **RAM** (256x8) for load/store instructions.

5.4 Load/Store Addressing

Loads and stores use a base register plus a small offset. The effective address is:

$$Address = BaseRegister + Offset$$

The offset is only 2 bits, but that's enough for testing memory behavior without making the instruction too big.

6. Instruction Format and Encodings

All instructions are **8 bits long**. I split them into four equal 2-bit fields:

$$Opcode(2) \mid Field1(2) \mid Field2(2) \mid Field3(2).$$

This made decoding much easier in Logisim since everything stays the same width.

6.1 Why 2 Bits for Everything?

- There are only four instructions, so the opcode only needs 2 bits.
- I have four registers, so each register field needs 2 bits.
- Offsets for load/store only need to be small (0–3), so 2 bits works fine.

6.2 Register Encodings

Register	Bits
R0	00
R1	01
R2	10
R3	11

6.3 Opcode Table

Instruction	Meaning	Opcode
ADD	Add two registers	00
SUB	Subtract two registers	01
LDR	Load from memory	10
STR	Store to memory	11

6.4 Instruction Formats

ADD Rd, Rn, Rm

$$00 \mid Rd \mid Rn \mid Rm$$

$$Rd \leftarrow Rn + Rm$$

SUB Rd, Rn, Rm

$$01 \mid Rd \mid Rn \mid Rm$$

$$Rd \leftarrow Rn - Rm$$

LDR Rt, offset(Rs)

$$10 \mid Rt \mid Rs \mid offset$$

$$Rt \leftarrow RAM[Rs + offset]$$

STR Rt, offset(Rs)

$$11 \mid Rt \mid Rs \mid offset$$

$$RAM[Rs + offset] \leftarrow Rt$$

7. Summary

Overall, the Silverback CPU is...

- four 8-bit general-purpose registers,
- ADD and SUB instructions on the ALU,
- load/store instructions with offset addressing,
- an 8-bit uniform instruction encoding,
- a working assembler and demo program,
- and a complete single-cycle datapath.

Even though the design is small, building it helped me understand how instructions actually move through the datapath and how control signals are generated.