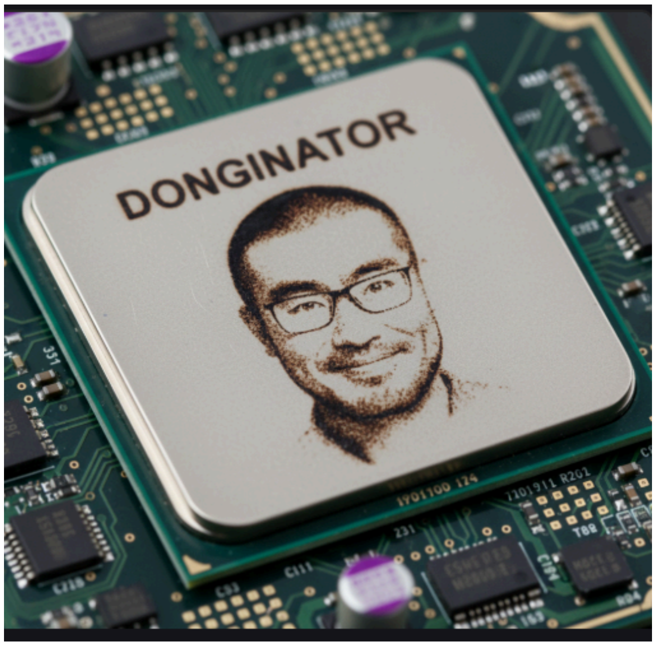
Donginator CPU User Manual

Alvaro Izquierdo & Adithya Ganesh

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



Donginator CPU is an 8-bit CPU with four general-purpose registers (X0–X3), two read ports, and one write port. Donginator CPU can perform addition and subtraction of two 8-bit numbers, with both operands coming from registers. It can also store data in memory using a small immediate number for the offset and load data from memory using a small immediate number as the offset.

Usage:

- Download the ‘Donginator CPU Package’ and ensure the following files are in your saved directory:
  - cpuproj.circ
  - translater.py
- Create and name your ARM64 assembly file to **program.s or program.txt** and place it in the same directory as the pre-existing program files.
- Open translater.py in VS Code (or any suitable IDE) and run it inside of a terminal window. This should create an output file called **image_file.txt** in the same directory as the program files.
- Open *cpuproj.circ* and locate the instruction memory in the Logisim circuit on the left side.
  - Right-click and use 'Load Image' to upload the **image_file.txt** for usage.
- After the instruction data is loaded, find the ‘Simulate’ tab in the top left of Logisim, and either manually tick through the program or enable ‘Auto-Tick’ to automatically run

through the program. You should see the registers and/or the memory update according to the instructions.

- To reset the circuit/simulation, find the ‘Simulate’ tab once again and use ‘Reset Simulation’ (CTRL R).

Architecture Description:

The CPU has four general-purpose registers that are referred to as X0, X1, X2, and X3. Each register can store an 8-bit number.

Instruction Syntax:

Our instruction set uses a simple register-based syntax similar to ARM-style notation, with registers X0–X3 and load/store using base+offset addressing. In the assembler implementation, these operations are encoded with the mnemonics ADDY, SUBBY, LDUR, and STUR, but conceptually they correspond to ADD, SUB, LDR, and STR as shown below.

- ADD Xm, Xn, Xd : *Xd = Xm + Xn*
  - (Assembler form: ADDY Xm Xn Xd)
- SUB Xm, Xn, Xd : *Xd = Xm - Xn*
  - (Assembler form: SUBBY Xm Xn Xd)
- LDR Xt Xn uimm2 : loads a byte from memory addressed by Xn + uimm2 into Xt.
  - (Assembler form: LDUR Xt Xn uimm2, where uimm2 is a 2-bit binary immediate such as 00, 01, 10, or 11.)
- STR Xt Xn uimm2 : stores a byte from Xn into memory addressed by Xn + uimm2.
  - (Assembler form: STUR Xt Xn uimm2, with the same 2-bit binary offset.)

ADD / SUB:

for register addition/subtraction:

| Opcode (2 bits) | | Reg1 (2 bits) | Reg2 / Rm (2 bits) | DstReg (2 bits)

The CPU uses a single 8-bit word for each arithmetic instruction. The top two bits of the byte are the opcode, the next two bits encode the first source register, the next two bits encode the second source register, and the last two bits encode the destination register. Both operands and the destination must be one of X0–X3.

LDR / STR:

| Opcode (2 bits) | DstReg / Xt (2 bits) | base / Reg1 (2 bits) | uimm2 (2 bits) |

For load and store, the instruction is also 8 bits. The top two bits encode the opcode (LDR or STR), the next two bits encode the target register Xt, the next two bits encode the base register, and the final two bits encode the 2-bit unsigned immediate offset uimm2 used in the effective address base + uimm2. When the opcode is LDR, DstReg is the destination register that receives the value loaded from memory at address Xn + uimm2. When the opcode is STR, Xt is the source register whose value is written to memory at address Xn + uimm2.

Binary Encodings:

Opcodes and registers are encoded as follows (2-bit fields):

- Opcode (bits 7-6):
    - 00 – SUB
    - 01 – LDR
    - 10 – STR
    - 11 – ADD
- Register codes (used in all register fields):
    - 00 – X0
    - 01 – X1
    - 10 – X2
    - 11 – X3
- Immediates
    - 00 – 0
    - 01 – 1
    - 10 – 2
    - 11 – 3

Here are some quick notes on the limitations of the 8-bit CPU:

- There are 4 general-purpose registers, ranging from X0–X3. Instructions such as ADD X5, X0, X1 will not perform correctly.
- Arithmetic operations only support register–register forms; there is no direct “ADD immediate” or “SUB immediate” instruction.
- The immediate offset in the data transfer instructions is only 2 bits, so uimm2 can range from 0 to 3. Larger offsets must be reached by adjusting the base register.

- All arithmetic and memory operations are 8-bit; values wrap around modulo 256, and no status flags (carry, overflow, etc.) are exposed.

Job Descriptions:

Alvaro's focus:

- Designed and wired the CPU datapath inside cpuproj.circ.
- Connected PC, register file, ALU, and memories.

Adithya's focus:

- Defined the 8-bit machine code format.
- Implemented the Python assembler (translater.py) to convert program.txt into image_file.txt for Logisim.
- Ensured correct instruction execution.

Joint work:

- Debugged CPU and assembler behavior.
- Refined the instruction set.
- Verified that CPU, assembler, and demo program functioned correctly as a complete system.