MCPU Extended Addressing

Design Notes

1 Context

The internet is full of amazing TTL and discrete transistor computer projects. I've always been fascinated by these seemingly impractical yet captivating displays of ground-up electronics design. This led me to consider how I might go about designing my own "home-made CPU."

One design that caught my eye was a pair of projects by Tim called the MCPU and LCPU. The first is a minimal CPU architecture for which he created a VHDL implementation, along with an assembler and emulator. The latter is a discrete component design using a variant of this CPU.

I found his work particularly well explained, making it an excellent starting point. The instruction set was exactly what I had in mind - small enough to keep complexity down, yet capable of running a "real" program. As for the physical design, which uses DTL (Diode Transistor Logic) with LEDs integrated into the logic, I consider it almost a work of art. Feel free to disagree.

1.1 Original MCPU

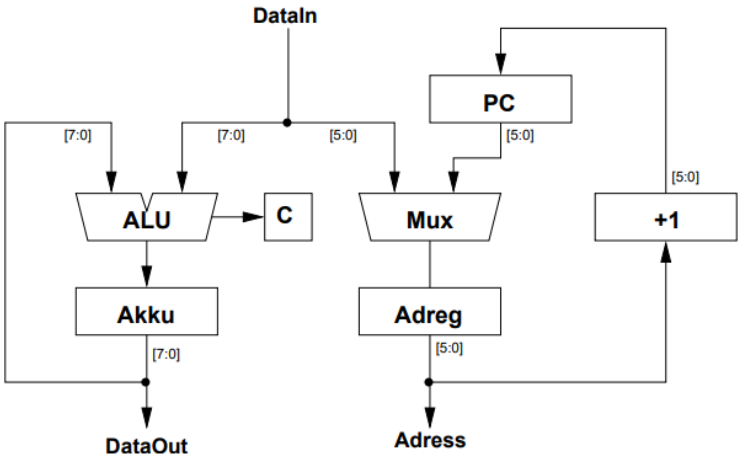

Fig. 1: Original MCPU architecture

With a 6 bit address register this design can address 64 bytes of memory. While this minimal CPU is intended as an educational tool primarily due to its limitations, the only real part holding it back is its memory size. There is only so much you can do with 64 bytes.

The main goal of this study is to improve on this initial design by extending the memory, while keeping the same instruction set and 8 bit data bus.

2 Architecture

In order to extend the addressable memory space, the address bus size obviously has to increase. Given that the data bus is 8 bits wide, it seems logical to increase the address bus to 8 bits as well. This would give 256 bytes of program memory which is already a big leap forward and plenty for actual real world applications.

For reference there are commercially available microprocessors with 512 bytes of program memory such as the ATTINY4. Keep in mind that this does not directly compare because it has 54 instructions which allows for more complex programs with the same amount of memory.

2.1 Indirect memory addressing

Increasing the address register size would require the instructions to be 10 bits (2 bit opcode + 8 bit address), breaking compatibility with standard 8 bit SRAM chips. To solve this problem the memory addressing mode is changed to **indirect addressing**, which allows the address register to be loaded from the data bus instead of the operand. Tim made a similar change for his LCPU design, allowing for the address register to be loaded from the accumulator.

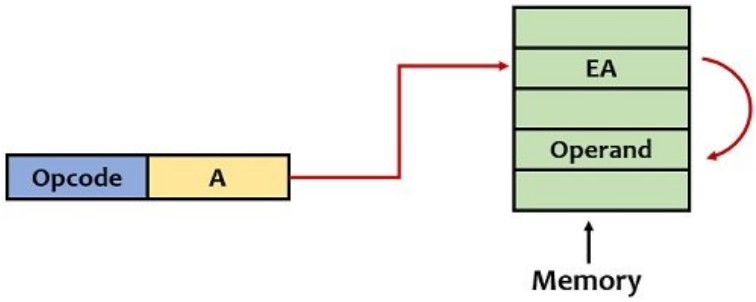

Fig. 2: Indirect addressing mode

The address in the operand (A) points to a vector table in memory containing the effective address (EA) which in turn points to the desired memory location. While the pointer address A is still 6 bit, the effective address EA is 8 bits and therefore allows to address a 256 byte SRAM.

Even though the vector table is limited to 64 bytes, with some creativity it is possible to fully utilize the memory (like incrementing the pointers in the vector table to work on arrays).

2.2 Memory banking

While 256 bytes is already a great improvement, a very simple modification requiring little to no logic resources allows us to achieve 512 bytes of addressable memory (2x256 bytes): a 9th bit will switch between 2 memory zones depending on the current status. This 9th bit doesn't require to increase the address register as it uses the status register, and thereby only requires a few logic gates.

Address	Area	Size
0x0000 0x00FF	Program memory	256 bytes
0x0100 0x010F	Vector table	64 bytes
0x0110 0x01FF	General purpose memory	192 bytes

2.3 VHDL implementation

The entity of the extended version of the MCPU is near identical to the basic design. The only difference is the address size which is 9 bits instead of 6.

```
entity mcpu_ext is
port (
    -- Control
    rst:    in  std_logic;  -- active low
    clk:    in  std_logic;
    -- SRAM interface
    data:   inout   std_logic_vector(7 downto 0);
    adress: out std_logic_vector(8 downto 0);
    oe:     out std_logic;  -- active low
    we:     out std_logic   -- active low
    );
end;
```

Fig. 3: MCPU-ext VHDL entity

This is where the similarities end however, as the VHDL implementation has been completely re-written from scratch. Instead of optimizing for as few logic cells as possible to fit in a 32 macro-cell CPLD, this new implementation aims for a better separation of sequential and combinational logic parts to facilitate the making of a physical TTL version. This also provides an explicit expression for the control signals.

In this spirit, all 3 registers are instances of a generic register and have the same interface as readily available register ICs. The accumulator register is reduced to 8 bits by separating the carry bit, leaving all 3 registers identical.

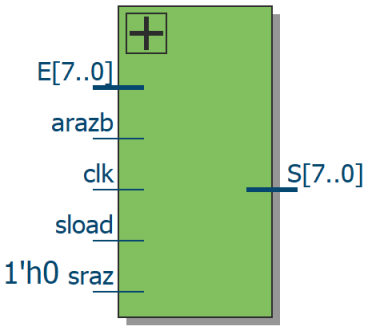

Fig. 4: Universal register

The VHDL code for the register is provided in the appendix 3.3.

2.3.1 SRAM trouble

If this design were to be implemented with TTL, or even discrete logic, the SRAM would certainly have to be a dedicated chip. Andrew Starr on hackaday created an amazing core memory module which could be a *somewhat practical* solution for such an amount of memory. But a more realistic approach would be the use of a chip like the REN-6116 from RENESAS.

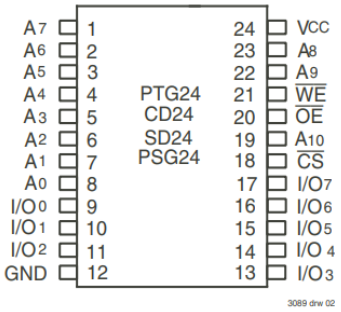

Fig. 5: MCPU-ext vhdl entity

With its 11 bit address bus, 2 extra bits can be connected to dip switches to enable multiple programs to be loaded and chosen from.

When implementing both the CPU and SRAM on an FPGA a problem arises. The SRAM above is non-synchronous, and therefore has no clock. But in most FPGA's the SRAM is synchronous and is edge triggered.

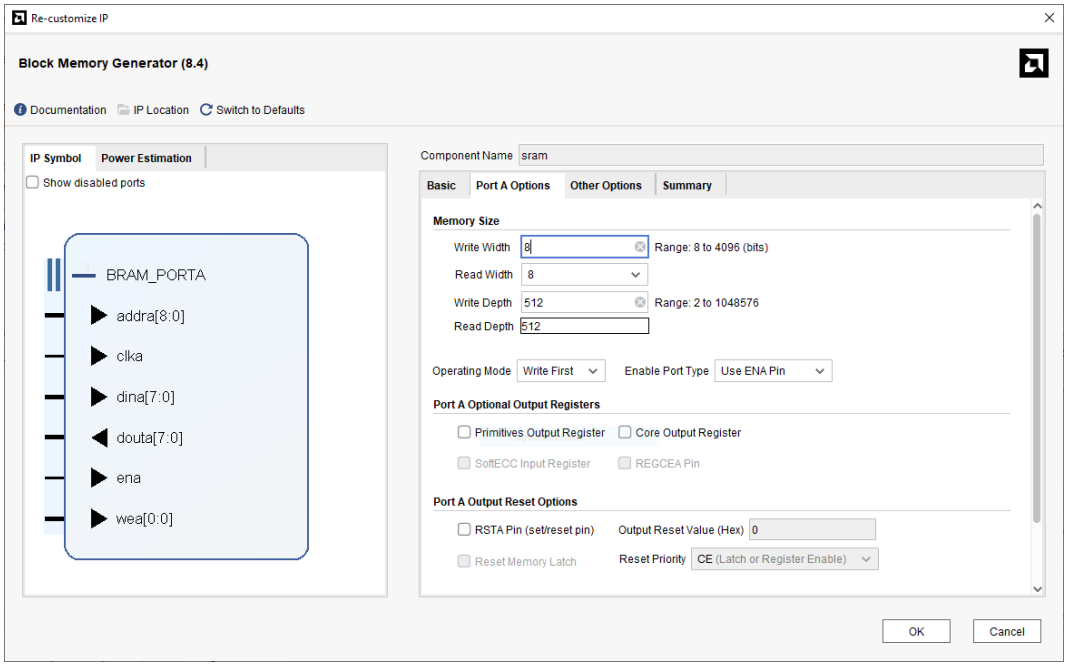

Fig. 6: SRAM IP

2.3.2 Simulation

In order to simulate the CPU, a testbench has to be created to connect our CPU to an SRAM from an IP as showed above, as well as clock and reset signals.

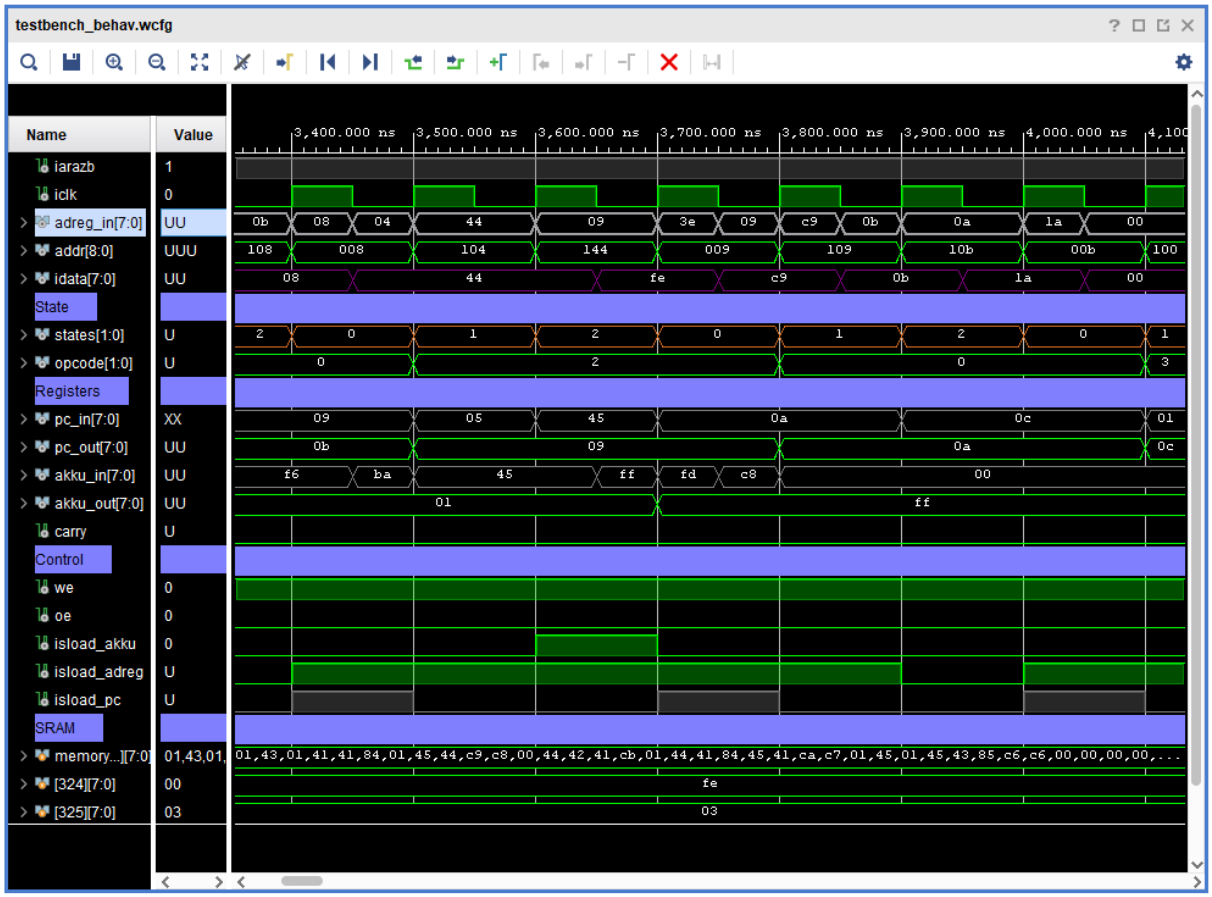

Fig. 7: Simulation

The resulting waveforms can than be inspected and compared to the software emulator output, which after some trial and error eventually match.

A thorough simulation would aim at "full coverage". In case of a CPU however this is nearly impossible due to the number of possibilities for a 512 byte rom (10^{1233} as far as my estimations go). The next best option is to verify all instructions, and for branching instructions make sure to cover all cases (we only have to wory about the JCC instruction).

3 Emulator

3.1 Converting to indirect addressing

The emulator was one of the easiest parts to convert to use indirect memory addressing. Given that the emulator uses an array as the memory, all that needs to be done is to replace to index from being the operand directly, to a "nested" index.

Original code :

```
mem[arg]
```

First I made an intermediate version which didn't have a "split" memory layout and was just the MCPU converted to indirect addressing, resulting in a 256 byte addressable memory space.

```
mem[mem[arg]]
```

To achieve the full 512 bytes, the MSB (Most Significant Bit) has to be set, which is the same as adding 256 like so :

```
mem[mem[arg+256]+256]
```

3.2 Test program

First 14 instructions of the prime test program (available in appendix 3.4) :

```
pc,ir,acc,cf:        00        0x01          0        0
pc,ir,acc,cf:        01        0x43          0        0
pc,ir,acc,cf:        02        0x01          2        0
pc,ir,acc,cf:        03        0x41          0        0
pc,ir,acc,cf:        04        0x41        255        0
pc,ir,acc,cf:        05        0x84        254        1
pc,ir,acc,cf:        06        0x01        254        1
pc,ir,acc,cf:        07        0x45          0        1
pc,ir,acc,cf:        08        0x44          3        0
pc,ir,acc,cf:        09        0xc9          1        1
pc,ir,acc,cf:        10        0xc8          1        0
pc,ir,acc,cf:        08        0x44          1        0
pc,ir,acc,cf:        09        0xc9        255        0
pc,ir,acc,cf:        11        0x00        255        0
pc,ir,acc,cf:        12        0x44          0        0
pc,ir,acc,cf:        13        0x42        254        0
```

Appendix

3.3 Universal register

```
library ieee;
use ieee.std_logic_1164.all;


entity reg_univ_n is
        generic ( n : natural := 4 ) ;
        port (
        clk, arazb : in std_logic ;
                sraz,sload : in std_logic ;
                E : std_logic_vector (n-1 downto 0) ;
                S : out std_logic_vector (n-1 downto 0) ) ;
end reg_univ_n ;

architecture synth of reg_univ_n is
        signal reg : std_logic_vector (n-1 downto 0);
begin

        process (clk,arazb)
        begin
                if arazb = '0' then reg <= (others => '0');
                elsif clk = '1' and clk'event then
                        if sraz = '1' then reg <= (others => '0') ;
                        elsif sload = '1' then reg <= E ;
                        end if ;
                end if ;
        end process ;

        S <= reg ;
end ;
```

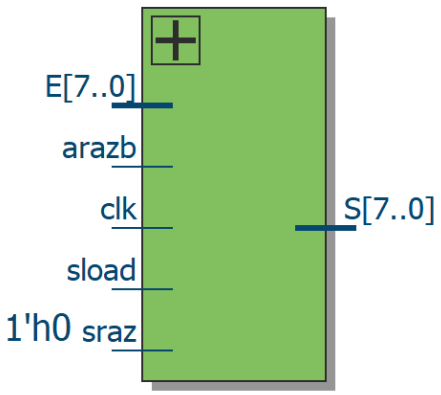

Fig. 8: Universal register

3.4 Prime number generator assembly code

```
        org 320
zero    dcb 0
allone  dcb 255
one     dcb 1
two     dcb 2
subs    dcb 0
number  dcb 3

        org 0

        lda two

start   lda allone
        add allone
        sta subs

loop    lda number
inner   add subs
        jcs inner

        sub subs
        add allone
        jcc noprime

        lda subs
        add allone
        sta subs

        add number
        add allone
        jcs loop

        lda number

noprime lda number
        add two
        sta number

        jmp start
```

3.5 VHDL Code

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity mcpu_ext is
port (
        -- Control
        rst:    in      std_logic;      -- active low
        clk:    in      std_logic;
        -- SRAM interface
        data:   inout   std_logic_vector(7 downto 0);
        adress: out     std_logic_vector(8 downto 0);
        oe:             out     std_logic;      -- active low
        we:             out     std_logic       -- active low
        );
end;

architecture ar of mcpu_ext is

component reg_univ_n is
generic ( n : natural := 4 ) ;
port (  clk, arazb : in std_logic ;
        sraz,sload : in std_logic ;
        E : std_logic_vector (n-1 downto 0) ;
        S : out std_logic_vector (n-1 downto 0) ) ;
end component ;

signal  states: std_logic_vector(1 downto 0);
signal  opcode: std_logic_vector(1 downto 0);

-- Data
signal  akku_in:        std_logic_vector(7 downto 0);
signal  akku_out:       std_logic_vector(7 downto 0);
signal  adreg_in:       std_logic_vector(7 downto 0);
signal  adreg_out:      std_logic_vector(7 downto 0);
signal  pc_in:          std_logic_vector(7 downto 0);
signal  pc_out:         std_logic_vector(7 downto 0);
signal  sum:            std_logic_vector(8 downto 0);
signal  carry:          std_logic;
signal  ista:       std_logic;

-- Control
signal isload_akku      : std_logic;
signal isload_adreg : std_logic;
signal isload_pc        : std_logic;

begin
        -- Registers
        reg_akku:reg_univ_n
        generic map( n => 8 )
        port map (
                clk     => clk,
                arazb   => rst,
                sraz    => '0',
                sload   => isload_akku,
```

```
        E               => akku_in,
        S               => akku_out
);

reg_adreg:reg_univ_n
generic map( n => 8 )
port map (
        clk     => clk,
        arazb   => rst,
        sraz    => '0',
        sload   => isload_adreg,
        E               => adreg_in,
        S               => adreg_out
);

reg_pc:reg_univ_n
generic map( n => 8 )
port map (
        clk     => clk,
        arazb   => rst,
        sraz    => '0',
        sload   => isload_pc,
        E               => pc_in,
        S               => pc_out
);

-- Address path
pc_in      <= adreg_out + 1;
isload_pc  <= states(1) nor states(0);
isload_adreg <= opcode(1) or opcode(0) or carry or isload_pc or
   states(0);
adreg_in   <=  pc_out when states(1) = '1' else
               data when states(0) = '1' else
               "00" & data(5 downto 0);

-- Data path
sum <= ("0" & akku_out) + ("0" & data);
akku_in <= sum(7 downto 0) when opcode = "10" else (akku_out nor
    data);
isload_akku <= states(1) when opcode(1) = '1' else '0';

process(clk,rst)
begin
        if (rst = '0') then
    states      <= "00";
    opcode      <= "00";
    carry   <= '0';
        elsif rising_edge(clk) then
    -- ALU / Data Path
    if (states(1) = '1') then
        case opcode is
            when "00"   => carry <= '0';     -- clear carry
            when "10"   => carry <= sum(8);  -- store carry
            when others => null;
        end case;
    end if;
```

```
            -- State machine
            if (isload_pc = '1') then
                states <= states(0) & '1';
                opcode <= not data(7 downto 6);
            else states <= states(0) & '0';
            end if;
                end if;
        end process;

        -- Output
        ista    <= not opcode(1) and opcode(0) and states(1);
        data    <= "ZZZZZZZZ" when ista = '0' else akku_out(7 downto 0);
        adress(7 downto 0)      <= adreg_out;
        adress(8) <= not isload_pc;
        oe              <= '1' when (rst='0' or ista = '1') else '0';
        we              <= '1' when (rst='0' or ista = '0') else '0';
end ar;
```

3.6 Assembler Python Code

```
def valof(arg):
    if arg in syms:
        arg = syms[arg]
    try:
        return int(arg)
    except:
        return 0

# pseudo ops
def org(arg):
    global pc
    pc = valof(arg)
def dcb(arg):
    global pc
    mem[pc] = valof(arg)
    pc += 1

# instruction set using indirect addressing
def nor(arg): dcb((valof(arg) & 0x3F) | 0x00)    # [00] akk nor data
def add(arg): dcb((valof(arg) & 0x3F) | 0x40)    # [01] akk + data
def sta(arg): dcb((valof(arg) & 0x3F) | 0x80)    # [10] store
def jcc(arg): dcb((valof(arg) & 0x3F) | 0xC0)    # [11] jmp if carry set

# combined instructions
def jmp(arg): jcc(arg); jcc(arg)
def lda(arg): nor('allone'); add(arg)
def sub(arg): nor('zero'); add(arg); add('one')
def out(arg): dcb(0xFF)
def jcs(arg):
    if str(pc+2) not in syms: syms[str(pc+2)] = pc+2
    jcc(str(pc+2))
    jcc(arg)

def build_vector_table():
    global pc, vector_table
    assert len(syms) < 64, "More␣than␣64␣elements␣in␣vector␣table␣({
       n_elements}␣elements)".format(n_elements=len(syms))
    vector_table = {}
    pc = 0
    for sym in syms:
        vector_table[sym] = pc
        mem[pc+256] = syms[sym]%256
        pc += 1

def assemble(input_file, output_file):
    # Load assembly file
    with open(input_file, 'r') as f:
        code = f.readlines()

    global syms, pc, mem
    syms = {}
    mem = 512 * [0]

    # 2-pass assembly
    for npass in range(2):
```

```
        pc = 0
        for line in code:
            fields = line.split()
            assert pc < 512, 'pc␣overflow␣:␣{}'.format(line)
            assert len(fields) < 4, 'Syntax␣error␣:␣{}'.format(line)
            if len(fields) > 0 and line.lstrip()[0] != ';':
                # Process labels
                if line.lstrip() == line:
                    if npass == 0:
                        syms[fields[0]] = pc
                    del fields[:1]
                # Assemble instruction
                f = globals()[fields[0]]
                f(fields[1])

        # Build vector table after first pass
        if npass == 0:
            build_vector_table()
            # Debug output
            print(vector_table)
            print(syms)
            syms = vector_table

    # Write generated machine code to output file
    with open(output_file, 'w') as f:
        s = '\n'
        f.write(s.join([format(x, '02X') for x in mem]))

import sys
if __name__ == "__main__":
    if len(sys.argv) != 3:
        print("Usage:␣python␣assembler.py␣<input_file>␣<output_file>")
        sys.exit(1)

    # Get arguments
    input_file = sys.argv[1]
    output_file = sys.argv[2]

    # Run assembler
    assemble(input_file, output_file)
```

3.7 Emulator Python Code

```
def emulate(input_file):
    with open(input_file, 'r') as f:
        mem = [int(x, 16) for x in f.readlines()]

    assert len(mem) == 512

    pc = 0
    acc = 0
    cf = 0

    for _ in range(cycles):
        ir = mem[pc]
        pc += 1
        op, arg = ir >> 6, ir & 0x3F

        # Display cpu state
        if cycles < 1000:
            print('pc,ir,acc,cf:', pc-1, hex(ir), acc, cf >> 8, sep='\t'
                )

        # Run instruction
        if op == 0:
            acc = (acc | mem[mem[arg+256]+256]) ^ 0xFF
        elif op == 1:
            acc += mem[mem[arg+256]+256]
            cf = acc & 0x100
            acc &= 0xFF
        elif op == 2:
            mem[mem[arg+256]+256] = acc
        else:
            if ir == 0xFF:
                print(acc)
            elif cf == 0:
                pc = mem[arg+256]
            cf = 0

import sys
if __name__ == "__main__":
    if len(sys.argv) < 2:
        print("Usage: python emu.py <input_file> <n_cycles>")
        sys.exit(1)

    try:
        cycles = int(sys.argv[2])
    except:
        cycles = 20

    input_file = sys.argv[1]

    emulate(input_file)
```