Designing A Soft Processor Using Verilog

Lucas Donovan
Milton and Doreen School of Engineering
Shippensburg University
Shippensburg, PA
ld0010@ship.edu

***Abstract*—This paper presents a soft processor designed using Verilog for the final project of both CMPE 420 and CMPE 330 at Shippensburg University. The processor is inspired by the Intel 4004—the first commercially produced microprocessor—however the design simplifies many aspects of the original architecture. The processor is implemented on a ZedBoard Zynq-7000, which has a 100 MHz clock frequency. The project utilizes the board's eight slide switches, eight LEDs, and three of its five available buttons for input and control, along with a two-digit seven-segment display module for output.**

***Index Terms*—Soft Processor, Instruction Set Architecture, Opcode, Zedboard, Verilog**

I. Introduction

A soft processor is a processor that is implemented entirely using logic synthesis. Designing a processor is a complicated task, so for the sake of this project the process was kept as simple as possible, while still retaining certain natural complexities involved in designing a processor. The goal was to use the eight input switches on the Zedboard as an instruction, and use the eight LEDs and two seven segment displays as output. The processor also has two reset buttons, one that resets all data, and one that only resets the ACC and PC (for keeping register values). One of the other buttons is also used as a clock. The processor also has a few important registers. The Program Counter (PC) is updated based on the input. It increases by one for most opcodes, however it can change to a desired value by using a jump operation. Since the operand is only 4 bits, there are only 16 different values for the PC (0-15). The Accumulator (ACC) holds intermediary values. Any operations, such as addition, will be performed to the ACC, and the results will be stored in the ACC. There are also 16 4-bit registers that can be used to store values. These general purpose registers can be accessed using the LOD and STR operations.

II. System Architecture

The processor follows a simple Von Neumann architecture. The instructions are 8 bits, split into a 4 bit opcode that defines one of 16 operations, and 4 bit operand that can define a register, immediate value, or memory address depending on the opcode. The system is accumulator-based, meaning that most operations are performed onto the accumulator register, ACC.

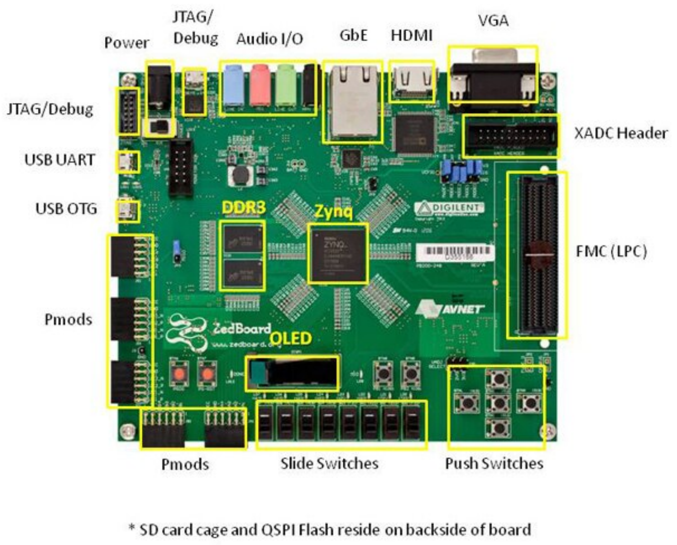

Fig. 1: Zedboard [1]

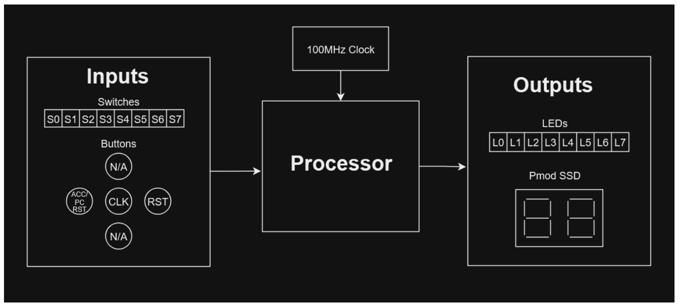

Fig. 2: System Overview

A. Control Unit

The control unit interprets the 4-bit opcode and manages:

- Instruction decoding
- Register file access
- Program counter updates
- Accumulator updates
- Condition code evaluation

The control unit is implemented as an if-else statement, with each if statement containing one of the three control signals (clock, reset, soft reset). Within the clock's if statement is the main logic, implemented with a case statement. The case statement checks the opcode, and performs the respective operation based on the input.

B. Input/Output System

The I/O system includes:

- 8 slide switches for instruction input
- 8 LEDs for status display
- 2-digit seven-segment display for output
- Button-based clock and reset control

The 8 switches designate the opcode and operand. On the board, the left 4 switches are the opcode, and the right 4 are the operand. The 8 LEDs represent the PC and ACC. The left 4 LEDs show the PC, and the right 4 show the ACC. The pmod seven segment display module used contains two digits. The left digit shows the current register number selected, and the right digit shows the value in that register. The board also has 5 push buttons, 3 of which are used for this processor. The buttons are arranged in a plus pattern, with the 3 middle buttons (left, center, right) in use and the 2 other buttons (top and bottom) are not in use. The center button is used to pulse the clock, the right button is used for total reset, and the left button is the soft reset, only clearing the PC and ACC.

C. Clock Management

The system uses a button-based clock with debouncing circuitry to ensure reliable operation. The button-based clock was chosen over a traditional uniform clock derived from the board's 100MHz clock for two simple reasons:

- The time between clock cycles will change based on the input instruction. For example, simply incrementing the ACC for 5 clock cycles will take a lot less time than a more complex set of instructions, purely due to the time it takes to change the input switches.
- Using a button to advance the clock simplifies debugging and demonstration, mainly due to the reason above.

D. Register File

The register file is implemented as an array of 16 4-bit registers:

```
reg [3:0] R [0:15];
```

Each of the 16 registers can hold a 4 bit value, corresponding to 0-F. This register array makes up the general purpose registers, used for storing and fetching values.

E. Other Registers

- 4-bit Accumulator (ACC)
- 4-bit Program Counter (PC)
- 2 Condition Codes
  - 1-bit Carry flag
  - 1-bit Zero flag

The two condition codes, carry and zero, are used for branching operations.

F. Instruction Set Architecture

This processor uses a 4 bit opcode and a single 4 bit operand. Having 4 bits for the opcode means that there can be 16 operations performed.

$b_0b_1b_2b_3$	$b_4b_5b_6b_7$
Opcode	Operand

TABLE I: Instruction Organization

Another possible route for the instruction could have been a 3 bit opcode and 5 bit operand, to operate on larger numbers. This halves the number of possible operations from 16 to 8, which is not ideal. This influenced the decision to split the input bits 50/50 between the opcode and operand, allowing for 16 different operations. These 16 operations are as follows:

- ROM: This opcode puts the processor into ROM mode, running the instructions in the ROM.
- LOD: This operation loads a value from the selected register (the operand) into the ACC.
- STR: This operation stores the value from the ACC into a selected register.
- ADD: This operation adds the value stored in the selected register (the operand) to the ACC. The result is stored in the ACC.
- SUB: This operation subtracts the value stored in the selected register (the operand) to the ACC. The result is stored in the ACC.
- AND: This operation performs a bit-wise AND with the value in the ACC and the operand.
- OR: This operation performs a bit-wise OR with the value in the ACC and the operand.
- LDI: This operation loads an immediate value (the operand) into the ACC.
- ADI: This operation adds an immediate value (the operand) to the the ACC.
- SUI: This operation subtracts an immediate value (the operand) from the ACC.
- LDI: This operation loads an immediate value (the operand) into the ACC.
- JMP: This operation performs an unconditional jump. The PC is set to the value of the operand.
- JNZ: This operation performs a jump (PC is set to operand) only if the zero bit is set to 0.
- JEZ: This operation performs a jump (PC is set to operand) only if the zero bit is set to 1.
- JNC: This operation performs a jump (PC is set to operand) only if the carry bit is set to 0.
- JC: This operation performs a jump (PC is set to operand) only if the carry bit is set to 1.
- HLT: This operation halts the execution, forbidding the clock to advance the PC until the reset is activated.

G. ROM

By creating a ROM, the processor can run small programs. Since the PC is only 4 bits, the ROM can only hold 16

Value	Opcode	Operation
0000	ROM	ROM Mode
0001	LOD	ACC = R[operand]
0010	STR	R[operand] = ACC
0011	ADD	ACC = ACC + R[operand]
0100	SUB	ACC = ACC - R[operand]
0101	AND	ACC = ACC & R[operand]
0110	OR	ACC = ACC | R[operand]
0111	LDI	ACC = operand
1000	ADI	ACC = ACC + operand
1001	SUI	ACC = ACC - operand
1010	JMP	PC = operand
1011	JNZ	PC = operand if ACC != 0
1100	JEZ	PC = operand if ACC = 0
1101	JNC	PC = operand if carry flag = 0
1110	JC	PC = operand if carry flag = 1
1111	HLT	Halt Execution

TABLE II: Opcodes

operations or ”lines of code”. The ROM module takes the current PC as input and outputs the instruction to be executed. It uses a simple case statement to decide which line of the program to run based on the input PC.

```
module Instruction_ROM (
    input [3:0] pc,
    output reg [7:0] instr
);
```

Example Rom Configuration - Counter:

```
0001_0000; // LOD 0 - ACC = R0
1000_0001; // ADI 1 - ACC = ACC + 1
0010_0000; // STR 0 - R0 = ACC
1011_0000; // JNZ 0 - PC = 0 if ACC != 0
1111_1111; // HLT F - Halt Execution
```

This program:

1) Loads the ACC with a value of 0
2) Increments the ACC by 1
3) Stores the new ACC value into register 0.
4) If the ACC is any value other than 0 (1-F), the PC jumps back to the second instruction. Otherwise, the execution stops.

In this case, the program will start off with PC = 0, ACC = 0, and R0 = 0. The program updates each clock cycle, producing an up counter in both the ACC and R0, updating every three clock cycles. The R0 register is used for easier output of the counter, instead of tracking the binary ACC on the board. The first values of each variable for the first ten clock cycles are shown in the table below.

Clock	PC	ACC	R0
0	0	0	0
1	1	1	0
2	2	1	1
3	3	1	1
4	0	1	1
5	1	2	1
6	2	2	2
7	3	2	2
8	0	2	2
9	1	3	2
10	2	3	3

TABLE III: Example Program: Counter

III. Conclusion

In conclusion, this paper outlines the design of a simple soft processor. The processor uses 8 bit instructions, splitting the instruction into a 4 bit opcode and a single 4 bit operand. If the opcode is all 0's, then the processor enters ROM mode, running the instructions in the ROM. Otherwise, the instruction is read from the input switches. The ROM can contain up to 16 total instructions, however these instructions can be branching instructions, possibly leading to programs that execute more than 16 instructions. One example is the counter program shown above. The processor contains 16 total opcodes, meaning 16 total operations. Of these opcodes, 6 instructions are register direct addressing, 3 are immediate instructions, and 5 are branching instructions. The other two are special instructions, one being the ROM mode instruction, the other being a HALT instruction. The HALT instruction restricts the program counter from advancing. The halt instruction is useful for stopping the execution of programs. The processor is clocked using a button on the Zedboard, with debouncing circuitry to ensure a stable clock input. Two other buttons are used for resetting the stored values. One reset button is used to reset all stored values, including PC, ACC, and all general purpose registers. The other reset button resets only the PC and ACC, keeping all stored values in the general purpose registers.

IV. References

[1] Avnet, ”ZedBoard,” Avnet Americas. [Online]. Available: https://www.avnet.com/americas/products/avnet-boards/avnet-board-families/zedboard/. Accessed: May 04, 2025.

APPENDIX

```
// Top Module
module Soft_Processor(
    output [7:0] LED,
    output [6:0] SEGMENT,
    output reg CTRL,
    input [7:0] SWITCH,
    input [2:0] BUTTON,
    input SYSCLK
);

    wire step;
    wire reset_all;
    wire pc_acc_reset;

    debounce STEP ( .clk(SYSCLK), .reset(1'b1), .button_in(BUTTON[0]),
    .button_out(step));
    debounce RESET (.clk(SYSCLK), .reset(1'b1), .button_in(BUTTON[1]),
    .button_out(reset_all));
    debounce RESET_PC_ACC (.clk(SYSCLK), .reset(1'b1), .button_in(BUTTON[2]),
    .button_out(pc_acc_reset));

    reg [3:0] ACC;
    reg [3:0] PC;
    reg [3:0] R [0:15];
    reg carry;

    wire [7:0] rom_instr;
    Instruction_ROM ROM (.pc(PC), .instr(rom_instr));

    wire use_rom = (SWITCH[7:4] == 4'h0);
    wire [7:0] current_instr = use_rom ? rom_instr : SWITCH;

    wire [3:0] opcode = current_instr[7:4];
    wire [3:0] operand = current_instr[3:0];

    integer count, desired;
    initial begin
        CTRL = 0;
        count = 0;
        desired = 49999;
        //desired = 5; // testbench
    end

    always @(posedge SYSCLK) begin
        if (count == desired) begin
            CTRL = ~CTRL;
            count = 0;
        end else begin
            count = count + 1;
        end
    end

    integer i;
    initial begin
        PC <= 4'h0;
        ACC <= 4'h0;
    end

    always @ (posedge step or posedge reset_all or posedge pc_acc_reset) begin
       if (reset_all) begin
            ACC <= 0;
            PC <= 0;
            carry <= 0;
            for (i = 0; i < 16; i = i + 1) begin
                R[i] <= 4'b0;
            end
        end else if (pc_acc_reset) begin
            ACC <= 0;
            PC  <= 0;
        end else if (step) begin
            case(opcode)
                4'h0: ;
                4'h1: ACC <= R[operand];
                4'h2: R[operand] <= ACC;
                4'h3: {carry, ACC} <= ACC + R[operand];
                4'h4: {carry, ACC} <= ACC - R[operand];
                4'h5: ACC <= ACC & R[operand];
                4'h6: ACC <= ACC | R[operand];
                4'h7: ACC <= operand;
                4'h8: {carry, ACC} <= ACC + operand;
                4'h9: {carry, ACC} <= ACC - operand;
                4'hA: PC <= operand;
                4'hB: if (!(ACC == 4'h0)) PC <= operand;
                4'hC: if (ACC == 4'h0) PC <= operand;
                4'hD: if (!carry) PC <= operand;
                4'hE: if (carry) PC <= operand;
                4'hF: ; // Halt
                default: ;
            endcase

            if (!(opcode == 4'hA || (opcode == 4'hB && !(ACC == 4'h0))
            || (opcode == 4'hC && (ACC == 4'h0)) || (opcode == 4'hD && !carry)
            || (opcode == 4'hE && carry) || opcode == 4'hF)) begin
                PC <= PC + 1;
            end

        end
    end

    wire [6:0] left;
    wire [6:0] right;
    SevenSegDisp LeftDisp (.segment(left), .value(operand));
    SevenSegDisp RightDisp (.segment(right), .value(R[operand]));

    wire [6:0] L_zero;
    wire [6:0] R_zero;
    SevenSegDisp Zero (.segment(L_zero), .value(0));
    SevenSegDisp R_Zero (.segment(R_zero), .value(R[0]));

    wire [6:0] left_seg = use_rom ? L_zero : left;
    wire [6:0] right_seg = use_rom ? R_zero : right;

    assign LED[7:4] = PC;
    assign LED[3:0] = ACC;
    assign SEGMENT = CTRL ? left_seg : right_seg;
endmodule

/*
Example Programs

Add Two Numbers
4'b0000: instr = 8'b0111_0101; // ACC = 5
4'b0001: instr = 8'b0011_1000; // ACC = ACC + 2
4'b0010: instr = 8'b1111_0000; // Stop Execution

Counter
// This counter does not restart at 1 when the registers are cleared on reset during execution
4'b0000: instr = 8'b1000_0000; // ACC = 0;
4'b0001: instr = 8'b0001_0000; // ACC = R0
4'b0010: instr = 8'b1000_0001; // ACC = ACC + 1
4'b0011: instr = 8'b1011_0000; // PC = 0 if ACC != 0
default: instr = 8'b1111_1111; // Halt Execution
```

```
// This counter restarts at 1 when the regitsters are cleared on reset
4'b0000: instr = 8'b1000_0001; // ACC = ACC + 1
4'b0001: instr = 8'b0010_0000; // R0 = ACC
4'b0010: instr = 8'b1011_0000; // PC = 0 if ACC != 0
default: instr = 8'b1111_1111; // Halt Execution

*/
module Instruction_ROM (
    input [3:0] pc,
    output reg [7:0] instr
);
    always @(*) begin
        case (pc)
            4'b0000: instr = 8'b0001_0000; // ACC = R0
            4'b0001: instr = 8'b1000_0001; // ACC = ACC + 1
            4'b0010: instr = 8'b0010_0000; // R0 = ACC
            4'b0011: instr = 8'b1011_0000; // PC = 0 if ACC != 0
            default: instr = 8'b1111_1111; // Halt Execution
        endcase
    end
endmodule

module SevenSegDisp(
    output reg [6:0] segment,
    input [3:0] value
);
    always @(*) begin
        case(value)
            4'b0000: segment = 7'b1111110;
            4'b0001: segment = 7'b0110000;
            4'b0010: segment = 7'b1101101;
            4'b0011: segment = 7'b1111001;
            4'b0100: segment = 7'b0110011;
            4'b0101: segment = 7'b1011011;
            4'b0110: segment = 7'b1011111;
            4'b0111: segment = 7'b1110000;
            4'b1000: segment = 7'b1111111;
            4'b1001: segment = 7'b1110011;
            4'b1010: segment = 7'b1110111;
            4'b1011: segment = 7'b0011111;
            4'b1100: segment = 7'b0001101;
            4'b1101: segment = 7'b0111101;
            4'b1110: segment = 7'b1001111;
            4'b1111: segment = 7'b1000111;
            default: segment = 7'b0000000;
        endcase
    end
endmodule

module debounce(
    input clk,
    input reset,        // Active-low
    input button_in,
    output button_out
);
    wire slow_clk_en;
    wire Q1, Q2, Q2_bar, Q0;

    clock_enable u1(
        .Clk_100M(clk),
        .reset_n(reset),
        .slow_clk_en(slow_clk_en)
    );

    my_dff_en d0(
        .DFF_CLOCK(clk),
        .reset_n(reset),
        .clock_enable(slow_clk_en),
        .D(button_in),
        .Q(Q0)
    );

    my_dff_en d1(
        .DFF_CLOCK(clk),
        .reset_n(reset),
        .clock_enable(slow_clk_en),
        .D(Q0),
        .Q(Q1)
    );

    my_dff_en d2(
        .DFF_CLOCK(clk),
        .reset_n(reset),
        .clock_enable(slow_clk_en),
        .D(Q1),
        .Q(Q2)
    );

    assign Q2_bar = ~Q2;
    assign button_out = Q1 & Q2_bar;
endmodule

module clock_enable(
    input Clk_100M,
    input reset_n,       // Active-low
    output slow_clk_en
);
    reg [26:0] counter;
    integer desired = 249999;
    //integer desired = 5;

    always @(posedge Clk_100M or negedge reset_n) begin
        if (!reset_n) begin
            counter <= 0;
        end else begin
            counter <= (counter >= desired) ? 0 : counter + 1;
        end
    end

    assign slow_clk_en = (counter == desired) ? 1'b1 : 1'b0;
endmodule

module my_dff_en(
    input DFF_CLOCK,
    input reset_n,       // Active-low
    input clock_enable,
    input D,
    output reg Q
);
    always @(posedge DFF_CLOCK or negedge reset_n) begin
        if (!reset_n) begin
            Q <= 1'b0;
        end else if (clock_enable) begin
            Q <= D;
        end
    end
endmodule
```

```
// Constraints
# INPUTS
# 8 Switches - SWITCH [7:0]
set_property IOSTANDARD LVCMOS18 [get_ports SWITCH[0]]
set_property PACKAGE_PIN F22 [get_ports SWITCH[0]]

set_property IOSTANDARD LVCMOS18 [get_ports SWITCH[1]]
set_property PACKAGE_PIN G22 [get_ports SWITCH[1]]

set_property IOSTANDARD LVCMOS18 [get_ports SWITCH[2]]
set_property PACKAGE_PIN H22 [get_ports SWITCH[2]]

set_property IOSTANDARD LVCMOS18 [get_ports SWITCH[3]]
set_property PACKAGE_PIN F21 [get_ports SWITCH[3]]

set_property IOSTANDARD LVCMOS18 [get_ports SWITCH[4]]
set_property PACKAGE_PIN H19 [get_ports SWITCH[4]]

set_property IOSTANDARD LVCMOS18 [get_ports SWITCH[5]]
set_property PACKAGE_PIN H18 [get_ports SWITCH[5]]

set_property IOSTANDARD LVCMOS18 [get_ports SWITCH[6]]
set_property PACKAGE_PIN H17 [get_ports SWITCH[6]]

set_property IOSTANDARD LVCMOS18 [get_ports SWITCH[7]]
set_property PACKAGE_PIN M15 [get_ports SWITCH[7]]


# 2 Buttons  - BUTTON [2:0]
set_property IOSTANDARD LVCMOS18 [get_ports BUTTON[0]]
set_property PACKAGE_PIN P16 [get_ports BUTTON[0]]

set_property IOSTANDARD LVCMOS18 [get_ports BUTTON[1]]
set_property PACKAGE_PIN R18 [get_ports BUTTON[1]]

set_property IOSTANDARD LVCMOS18 [get_ports BUTTON[2]]
set_property PACKAGE_PIN N15 [get_ports BUTTON[2]]


# OUTPUTS
# 8 LEDs - LED [7:0]
set_property IOSTANDARD LVCMOS18 [get_ports LED[0]]
set_property PACKAGE_PIN T22 [get_ports LED[0]]

set_property IOSTANDARD LVCMOS18 [get_ports LED[1]]
set_property PACKAGE_PIN T21 [get_ports LED[1]]

set_property IOSTANDARD LVCMOS18 [get_ports LED[2]]
set_property PACKAGE_PIN U22 [get_ports LED[2]]

set_property IOSTANDARD LVCMOS18 [get_ports LED[3]]
set_property PACKAGE_PIN U21 [get_ports LED[3]]

set_property IOSTANDARD LVCMOS18 [get_ports LED[4]]
set_property PACKAGE_PIN V22 [get_ports LED[4]]

set_property IOSTANDARD LVCMOS18 [get_ports LED[5]]
set_property PACKAGE_PIN W22 [get_ports LED[5]]

set_property IOSTANDARD LVCMOS18 [get_ports LED[6]]
set_property PACKAGE_PIN U19 [get_ports LED[6]]

set_property IOSTANDARD LVCMOS18 [get_ports LED[7]]
set_property PACKAGE_PIN U14 [get_ports LED[7]]


# 2 Seven Segment Displays
set_property IOSTANDARD LVCMOS18 [get_ports SEGMENT[6]]
set_property PACKAGE_PIN Y11 [get_ports SEGMENT[6]]

set_property IOSTANDARD LVCMOS18 [get_ports SEGMENT[5]]
set_property PACKAGE_PIN AA11 [get_ports SEGMENT[5]]

set_property IOSTANDARD LVCMOS18 [get_ports SEGMENT[4]]
set_property PACKAGE_PIN Y10 [get_ports SEGMENT[4]]

set_property IOSTANDARD LVCMOS18 [get_ports SEGMENT[3]]
set_property PACKAGE_PIN AA9 [get_ports SEGMENT[3]]

set_property IOSTANDARD LVCMOS18 [get_ports SEGMENT[2]]
set_property PACKAGE_PIN W12 [get_ports SEGMENT[2]]

set_property IOSTANDARD LVCMOS18 [get_ports SEGMENT[1]]
set_property PACKAGE_PIN W11 [get_ports SEGMENT[1]]

set_property IOSTANDARD LVCMOS18 [get_ports SEGMENT[0]]
set_property PACKAGE_PIN V10 [get_ports SEGMENT[0]]

set_property IOSTANDARD LVCMOS18 [get_ports CTRL]
set_property PACKAGE_PIN W8 [get_ports CTRL]

# create clock
set_property PACKAGE_PIN Y9 [get_ports SYSCLK]
set_property IOSTANDARD LVCMOS18 [get_ports SYSCLK]
create_clock -name sys_clk_pin -period 10 -waveform [0 5] [get_ports SYSCLK]
```