Andrew Friedman
Lab 7 Report
ECE 2031 CS
06 July 2023

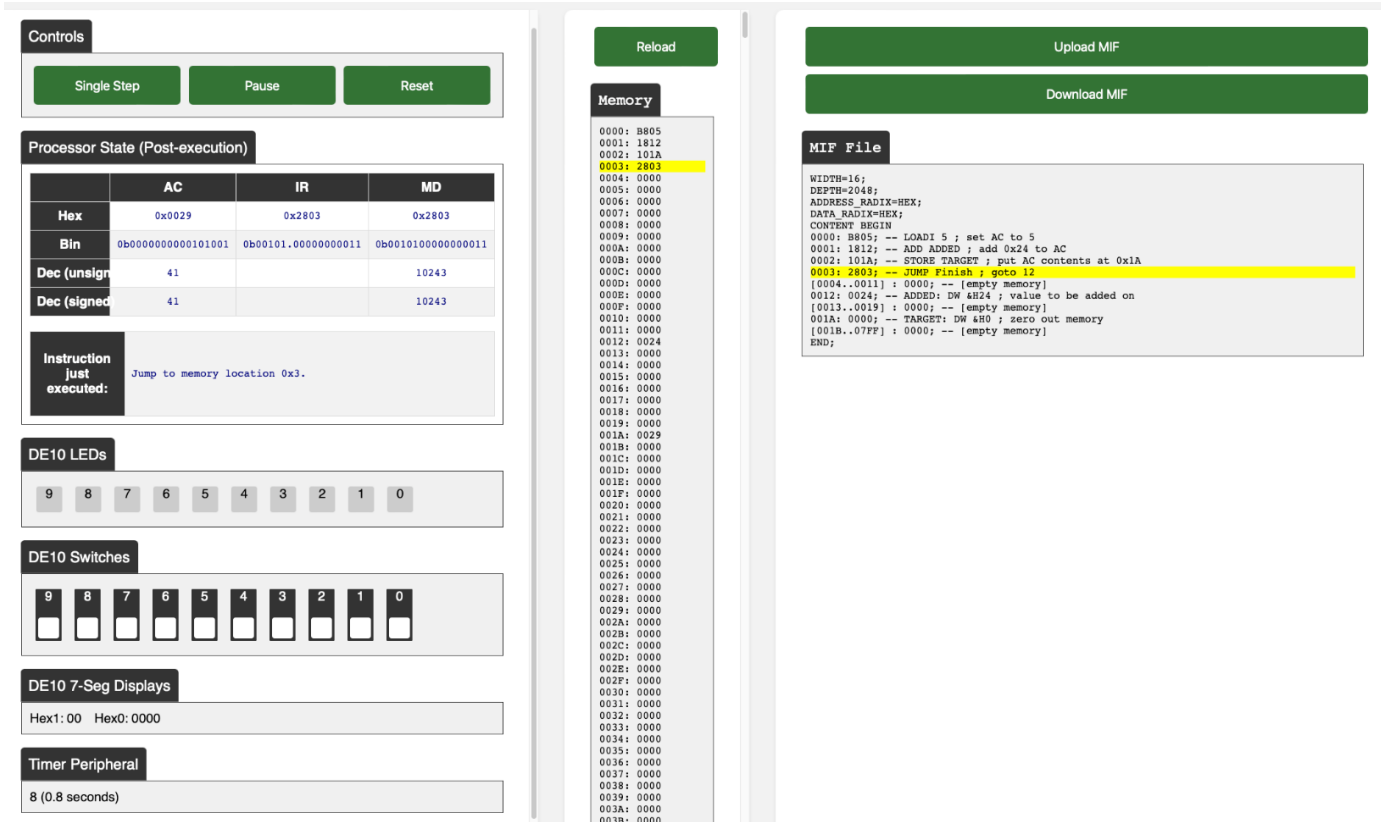

Figure 1. SCOMP online simulator screenshot within the custom infinite loop, displaying final AC, IR, and MD states.

```
1   ; step 6 - pre.asm
2   ; ASSEMBLY CODE DEMONSTRATING LOAD, ADD, STORE OPERATIONS, AND AN INFINITE LOOP INITIATION
3   ; Andrew Friedman
4   ; ECE 2031 CS
5   ; 07/06/2023
6
7   ; init program
8   ORG 0
9       LOADI   5               ; set AC to 5
10      ADD ADDED               ; add 0x24 to AC
11      STORE   TARGET          ; put AC contents at 0x1A
12  Finish:                     ; infinite loop
13      JUMP    Finish          ; goto 12
14
15  ; pre-set values
16  ORG &H012
17      ADDED:      DW  &H24    ; value to be added on
18  ORG &H01A                   ; memory destination
19      TARGET:     DW  &H0     ; zero out memory
```

Figure 2. Assembly code demonstrating load, add, store operations, and an infinite loop initiation.

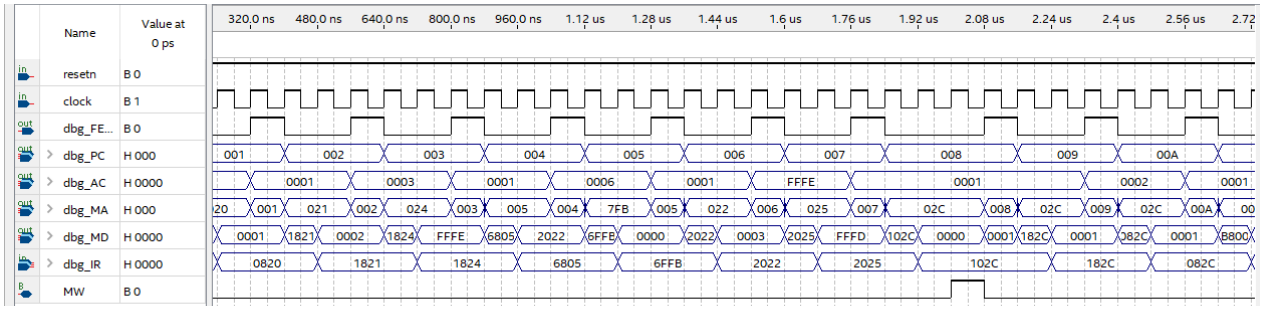

Figure 3. Simulation output demonstrating successful execution of the updated assembly program, with emphasis on the correct operation of the newly added SUB instruction.

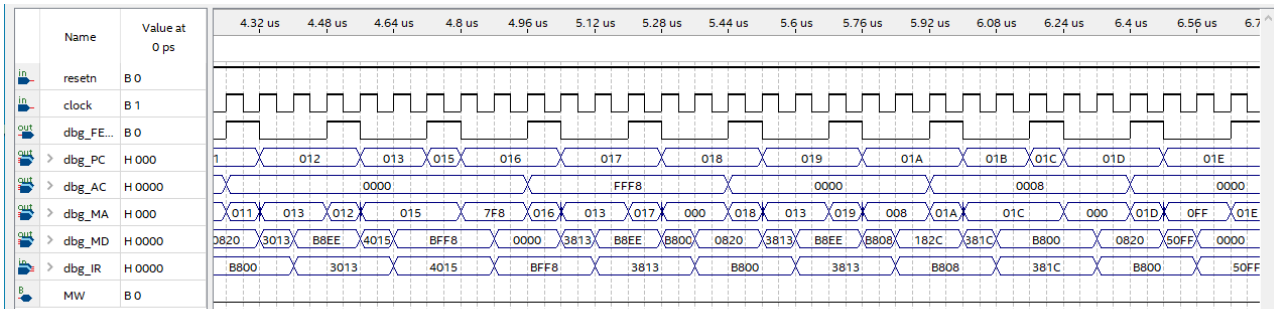

Figure 4. Simulation output demonstrating successful execution of the updated assembly program, with emphasis on the correct operation of the newly added JPOS instruction.

```
; Arithmetic.asm
; ITERATIVE MULTIPLICATION AND SUBTRACTION OPERATIONS ON AN INITIAL VALUE, WITH THE FINAL RESULT STORED IN AC
; Andrew Friedman
; ECE 2031 CS
; 07/06/2023

; init program
ORG 0

; double value in init & check if over max
Double:
    LOAD    Curr            ; load current value
    SHIFT   1               ; *= 2
    STORE   Curr            ; cache result
    SUB     Max             ; -= 1200
    JNEG    Double          ; not over so not finished double
    JZERO   Double          ; not over so not finished double

; subtract 50 from init & check if under min
Subtract:
    LOAD    Curr            ; load current value
    SUB     Fifty           ; -= 50
    STORE   Curr            ; cache result
    SUB     Min             ; -= 1196
    JPOS    Subtract        ; not under so not finished sub
    JZERO   Subtract        ; not under so not finished sub
    LOAD    Curr            ; load final result

Finish:                     ; infinite loop
    JUMP    Finish          ; goto 27

; values
Curr:   DW  162
; static values
Fifty:  DW  50
Max:    DW  1200
Min:    DW  1196
```

Figure 5. Assembly code ('Arithmetic.asm') performing iterative multiplication and subtraction operations on an initial value, with the final result stored in AC.

APPENDIX A
VHDL CODE HIGHLIGHTING THE NEWLY IMPLEMENTED SUB AND JPOS INSTRUCTIONS IN THE SCOMP PROCESSOR

```
-- SCOMP.vhd
-- This VHDL defines a simple 16-bit processor that is easy to understand and modify.
-- Andrew Friedman
-- ECE 2031 CS
-- 07/06/2023

library altera_mf;
library lpm;
library ieee;

use altera_mf.altera_mf_components.all;
use lpm.lpm_components.all;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_arith.all;


entity SCOMP is
	port(
		clock     : in    std_logic;
		resetn    : in    std_logic;
		IO_WRITE  : out   std_logic;
		IO_CYCLE  : out   std_logic;
		IO_ADDR   : out   std_logic_vector(10 downto 0);
		IO_DATA   : inout std_logic_vector(15 downto 0);
		dbg_FETCH : out   std_logic;
		dbg_AC    : out   std_logic_vector(15 downto 0);
		dbg_PC    : out   std_logic_vector(10 downto 0);
		dbg_MA    : out   std_logic_vector(10 downto 0);
		dbg_MD    : out   std_logic_vector(15 downto 0);
		dbg_IR    : out   std_logic_vector(15 downto 0)
	);
end SCOMP;

architecture a of SCOMP is
	type state_type is (
		init, fetch, decode, ex_nop,
		ex_load, ex_store, ex_store2, ex_iload, ex_istore, ex_istore2, ex_loadi,
		ex_add, ex_addi, ex_sub,
		ex_jump, ex_jneg, ex_jzero, ex_jpos,
		ex_return, ex_call,
		ex_and, ex_or, ex_xor, ex_shift,
		ex_in, ex_in2, ex_out, ex_out2
	);
```

```
	type stack_type is array (0 to 9) of std_logic_vector(10 downto 0);

	signal state       : state_type;
	signal AC          : std_logic_vector(15 downto 0);
	signal AC_shifted  : std_logic_vector(15 downto 0);
	signal PC_stack    : stack_type;
	signal IR          : std_logic_vector(15 downto 0);
	signal mem_data    : std_logic_vector(15 downto 0);
	signal PC          : std_logic_vector(10 downto 0);
	signal next_mem_addr    : std_logic_vector(10 downto 0);
	signal operand     : std_logic_vector(10 downto 0);
	signal MW          : std_logic;
	signal IO_WRITE_int : std_logic;


begin
	-- use altsyncram component for unified program and data memory
	altsyncram_component : altsyncram
	GENERIC MAP (
		numwords_a => 2048,
		widthad_a => 11,
		width_a => 16,
		init_file => "Arithmetic.mif",
		intended_device_family => "CYCLONE V",
		clock_enable_input_a => "BYPASS",
		clock_enable_output_a => "BYPASS",
		lpm_hint => "ENABLE_RUNTIME_MOD=NO",
		lpm_type => "altsyncram",
		operation_mode => "SINGLE_PORT",
		outdata_aclr_a => "NONE",
		outdata_reg_a => "UNREGISTERED",
		power_up_uninitialized => "FALSE",
		read_during_write_mode_port_a => "NEW_DATA_NO_NBE_READ",
		width_byteena_a => 1
	)
	PORT MAP (
		wren_a    => MW,
		clock0    => clock,
		address_a => next_mem_addr,
		data_a    => AC,
		q_a       => mem_data
	);
```

```
-- use lpm function to shift AC
shifter: lpm_clshift
generic map (
        lpm_width     => 16,
        lpm_widthdist => 4,
        lpm_shifttype => "arithmetic"
)
port map (
        data      => AC,
        distance  => IR(3 downto 0),
        direction => IR(4),
        result    => AC_shifted
);

-- Memory address comes from PC during fetch, otherwise from operand
with state select next_mem_addr <=
        PC when fetch,
        operand when others;

-- This makes the operand available immediately after fetch, and also
-- handles indirect addressing of iload and istore
with state select operand <=
        mem_data(10 downto 0) when decode,
        mem_data(10 downto 0) when ex_iload,
        mem_data(10 downto 0) when ex_istore2,
        IR(10 downto 0) when others;

-- use lpm function to drive i/o bus
io_bus: lpm_bustri
generic map (
        lpm_width => 16
)
port map (
        data     => AC,
        enabledt => IO_WRITE_int,
        tridata  => IO_DATA
);

IO_ADDR  <= IR(10 downto 0);
IO_WRITE <= IO_WRITE_int;

process (clock, resetn)
begin
        if (resetn = '0') then          -- Active-low asynchronous reset
```

```
                state <= init;
            elsif (rising_edge(clock)) then
                case state is
                    when init =>
                        MW       <= '0';          -- clear memory write flag
                        PC       <= "00000000000"; -- reset PC to the beginning of
memory, address 0x000
                        AC       <= x"0000";      -- clear AC register
                        IO_WRITE_int <= '0';       -- don't drive IO
                        state    <= fetch;        -- start fetch-decode-execute cycle

                    when fetch =>
                        IO_WRITE_int <= '0';   -- lower IO_WRITE after an out
                        PC       <= PC + 1;   -- increment PC to next instruction
address
                        state    <= decode;

                    when decode =>
                        IR   <= mem_data;         -- latch instruction into the IR
                        case mem_data(15 downto 11) is -- opcode is top 5 bits of
instruction
                            when "00000" =>      -- no operation (nop)
                                state <= ex_nop;
                            when "00001" =>      -- load
                                state <= ex_load;
                            when "00010" =>      -- store
                                state <= ex_store;
                            when "00011" =>      -- add
                                state <= ex_add;
                            when "00100" =>      -- sub
                                state <= ex_sub;
                            when "00101" =>      -- jump
                                state <= ex_jump;
                            when "00110" =>      -- jneg
                                state <= ex_jneg;
                            when "00111" =>      -- jpos
                                state <= ex_jpos;
                            when "01000" =>      -- jzero
                                state <= ex_jzero;
                            when "01001" =>      -- and
                                state <= ex_and;
                            when "01010" =>      -- or
                                state <= ex_or;
                            when "01011" =>      -- xor
```

```
                        state <= ex_xor;
                    when "01100" =>       -- shift
                        state <= ex_shift;
                    when "01101" =>       -- addi
                        state <= ex_addi;
                    when "01111" =>       -- istore
                        state <= ex_istore;
                    when "01110" =>       -- iload
                        state <= ex_iload;
                    when "10000" =>       -- call
                        state <= ex_call;
                    when "10001" =>       -- return
                        state <= ex_return;
                    when "10010" =>       -- in
                        state <= ex_in;
                    when "10011" =>       -- out
                        state <= ex_out;
                        IO_WRITE_int <= '1'; -- raise IO_WRITE
                    when "10111" =>       -- loadi
                        state <= ex_loadi;
                    when others =>
                        state <= ex_nop;   -- invalid opcodes default
to nop
                end case;

            when ex_nop =>
                state <= fetch;

            when ex_load =>
                AC    <= mem_data;        -- latch data from mem_data
(memory contents) to AC
                state <= fetch;

            when ex_store =>
                MW    <= '1';             -- drop MW to end write cycle
                state <= ex_store2;

            when ex_store2 =>
                MW    <= '0';             -- drop MW to end write cycle
                state <= fetch;

            when ex_add =>
                AC    <= AC + mem_data;   -- addition
                state <= fetch;
```

```
when ex_sub =>
        AC    <= AC - mem_data;   -- subtraction
        state <= fetch;

when ex_jump =>
        PC    <= operand; -- overwrite PC with new address
        state <= fetch;

when ex_jneg =>
        if (AC(15) = '1') then
                PC    <= operand;      -- Change the program
counter to the operand
        end if;
        state <= fetch;

when ex_jpos =>
        if (AC(15) = '0' and AC /= x"0000") then
                PC    <= operand;      -- Change the program
counter to the operand
        end if;
        state <= fetch;

when ex_jzero =>
        if (AC = x"0000") then
                PC    <= operand;
        end if;
        state <= fetch;

when ex_and =>
        AC    <= AC and mem_data;   -- logical bitwise AND
        state <= fetch;

when ex_or =>
        AC    <= AC or mem_data;
        state <= fetch;

when ex_xor =>
        AC    <= AC xor mem_data;
        state <= fetch;

when ex_shift =>
        AC    <= AC_shifted;
        state <= fetch;
```

```
when ex_addi =>
        -- sign extension
        AC    <= AC + (IR(10) & IR(10) & IR(10) &
         IR(10) & IR(10) & IR(10 downto 0));
        state <= fetch;

when ex_call =>
        for i in 0 to 8 loop
                PC_stack(i + 1) <= PC_stack(i);
        end loop;
        PC_stack(0) <= PC;
        PC          <= operand;
        state       <= fetch;

when ex_return =>
        for i in 0 to 8 loop
                PC_stack(i) <= PC_stack(i + 1);
        end loop;
        PC          <= PC_stack(0);
        state       <= fetch;

when ex_iload =>
        -- indirect addressing is handled in next_mem_addr
assignment.
        state       <= ex_load;

when ex_istore =>
        MW          <= '1';
        state       <= ex_istore2;

when ex_istore2 =>
        MW          <= '0';
        state       <= fetch;

when ex_in =>
        IO_CYCLE <= '1';
        state <= ex_in2;

when ex_in2 =>
        IO_CYCLE <= '0';
        AC <= IO_DATA;
        state <= fetch;
```

```
                        when ex_out =>
                                IO_CYCLE <= '1';
                                state <= ex_out2;

                        when ex_out2 =>
                                IO_CYCLE <= '0';
                                state <= fetch;

                        when ex_loadi =>
                                AC    <= (IR(10) & IR(10) & IR(10) &
                                 IR(10) & IR(10) & IR(10 downto 0));
                                state <= fetch;

                        when others =>
                                state <= init;          -- if an invalid state is reached, reset

                end case;
        end if;
    end process;

    dbg_FETCH <= '1' when state = fetch else '0';
    dbg_PC  <= PC;
    dbg_AC  <= AC;
    dbg_MA  <= next_mem_addr;
    dbg_MD  <= mem_data;
    dbg_IR  <= IR;

end a;
```