Rudra Goel
Lab 07 Report
ECE 2031 L10
10 October 2024

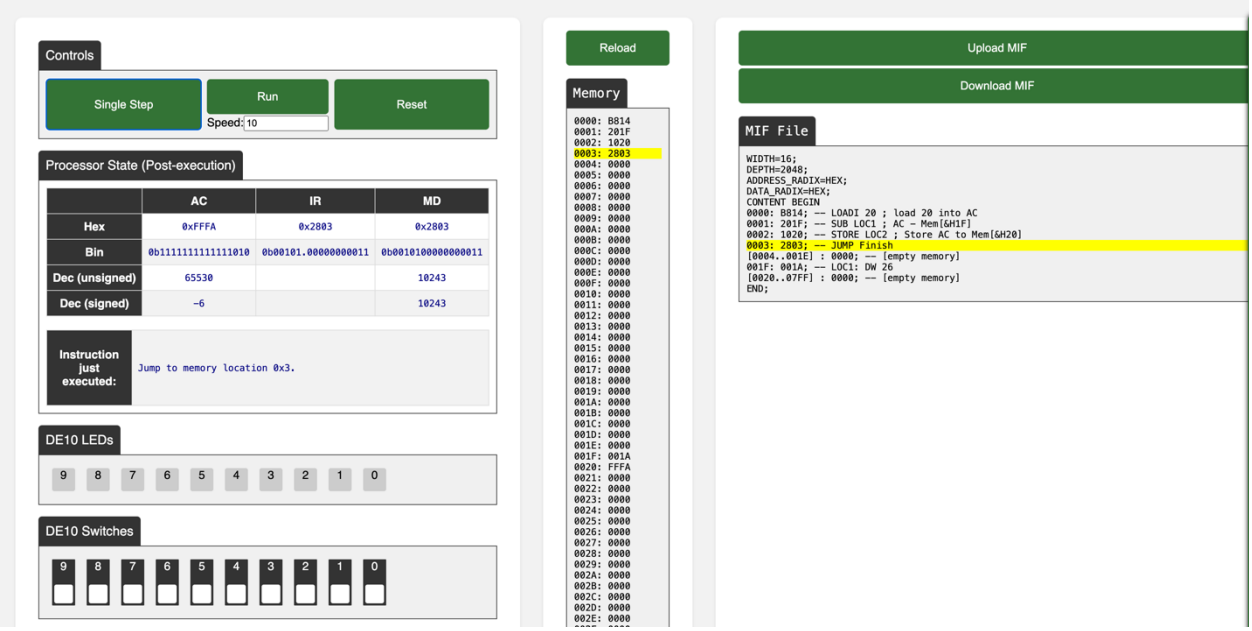

Figure 1. Capture of assembly code running on SCOMP processor simulator. This assembly code loads the value 20 into the accumulator, subtracts 26 from it (stored at memory location 0x1F), stores the result in memory location 0x20, and then enters an infinite loop called 'Finish' to mark the end of the program.

```
; Prelab Step 6 Assembly code to compute 20 - 26 in SCOMP arch.
; Author: Rudra Goel
; Date: 10/10/2024
ORG 0

     LOADI     20         ; load 20 into AC
     SUB       LOC1 ; AC - Mem[&H1F]
     STORE     LOC2 ; Store AC to Mem[&H20]


Finish:
     JUMP Finish

ORG  &H1F
LOC1:     DW   26

ORG  &H20
LOC2:     DW   0
```

Figure 2. Assembly code used to subtract 26 from 20 and store results in memory location 0x20. This code enters an infinite loop with the command 'JUMP Finish.'

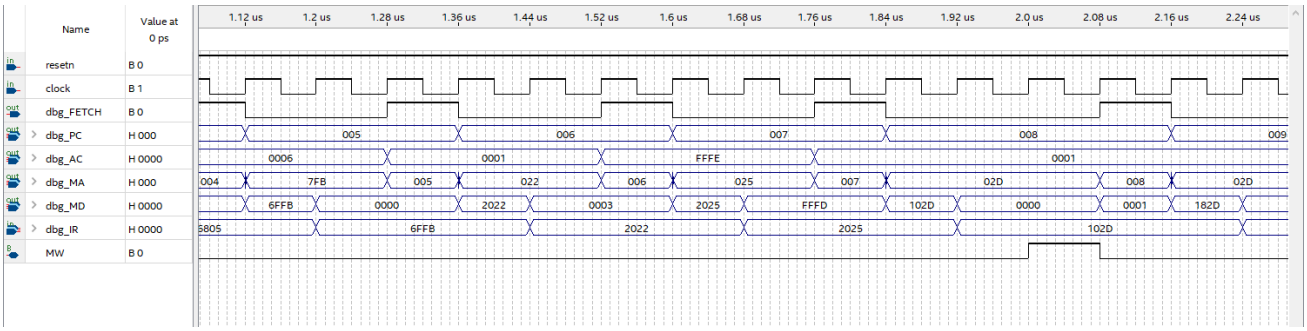

Figure 3. Capture of fixed waveform for SCOMP Processor that correctly implements op codes 'SUB.' Correct implementation is seen when signal 'dbg_AC' (representing the accumulator) goes from value '0001' to value 'FFFE' indicating subtraction by 5 has occurred in 2's complement form.

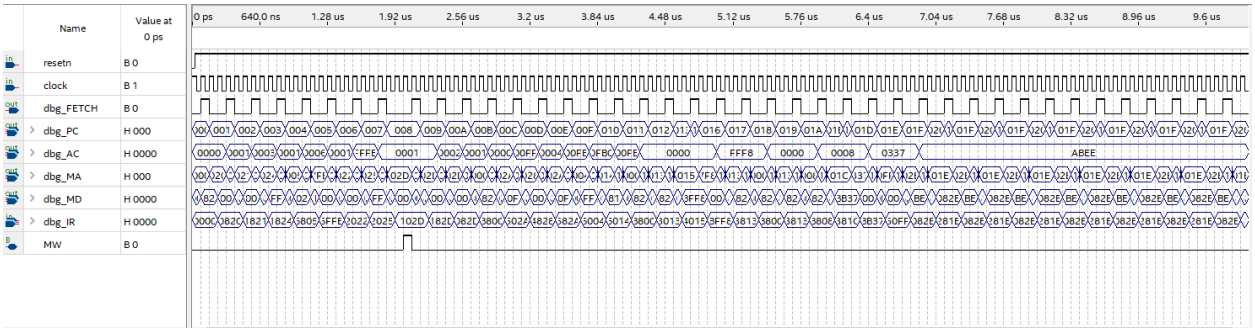

Figure 4. Full waveform capture of SCOMP Processor illustrating correct implementation of 'JPOS' op code. Correct implementation is seen at the end when signal 'dbg_AC' (representing the accumulator) goes from '0337,' a positive value, to 'ABEE' indicating a jump has occurred when the accumulator is positive (JPOS functionality) where it then loads value 'ABEE' from memory into the accumulator.

```
; Nibble Difference Calculator for 16-Bit Numbers
; Author: Rudra Goel
; Date: 10/10/2024

ORG 0
     LOAD Value
    AND   LOW_ONES       ; AC has 0 for 15-4 and nible for rest
    STORE LOW_NIBLE      ; LOW_NIBLE now has bits 3-0 of value
    LOAD Value
    SHIFT -12            ; shift 12 bits right to get
                         ; highest nible
    AND LOW_ONES         ; AC has bits 0-3 have highest nible
    STORE HIGH_NIBLE
    SUB LOW_NIBLE        ; high nible - low nible
    JPOS HighBigger
    LOAD LOW_NIBLE       ; low nible is bigger since AC is
negative since it didn't jump
    STORE RESULT
    JUMP End

HighBigger:
     LOAD HIGH_NIBLE
    STORE RESULT
    JUMP End

End:
     JUMP End


Value:          DW   &HFF11
LOW_ONES: DW    &H000F
LOW_NIBLE:      DW   &H0000
HIGH_NIBLE:     DW   &H0000
RESULT:         DW   &H0000
```

Figure 5. Assembly code for outputting greatest nibble between most-significant-nibble and least-significant-nibble in a 16-bit number, specified by label ‘Value,’ for the SCOMP Processor. Stores the result in memory location specified by the label ‘RESULT.’

Appendix A
VHDL Implementing Behavior Of SCOMP Processor Architecture

```
-- SCOMP, the Simple Computer.
-- This VHDL defines a simple 16-bit processor that is
-- easy to understand and modify.
-- Updated 2021-06-22

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
     );

     -- custom type for the call stack
```

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

     -- internal signals
     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 => "SimpleDemo.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_reg_a => "UNREGISTERED",
          outdata_aclr_a => "NONE",
          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 tri-state driver 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
     -- Active-low asynchronous reset
     if (resetn = '0') then
```

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

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

               when decode =>
-- latch all 16 bits of instruction into the IR
                    IR    <= mem_data;
-- opcode is top 5 bits of instruction
                    case mem_data(15 downto 11) is
-- no operation (nop)
                         when "00000" =>
                              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;
                    -- raise IO_WRITE
                         IO_WRITE_int <= '1';
                    when "10111" =>        -- loadi
                         state <= ex_loadi;
                    when others =>
               -- invalid opcodes don't execute
                         state <= fetch;
               end case;

               when ex_nop =>
                    state <= fetch;

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

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

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

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

                when ex_sub =>
                     AC   <= AC - mem_data;        -- sub
                     state     <= fetch;

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

                when ex_jneg =>
-- checks MSB of AC and if it is 1 --> number is negative
                     if (AC(15) = '1') then
-- Change the program counter to the operand
                          PC    <= operand;
                     end if;
                     state <= fetch;

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

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

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

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

                when ex_xor =>
                     AC    <= AC xor mem_data;
                     state <= fetch;
     -- shift is accomplished with a dedicated shifter
                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 =>
    -- if an invalid state is reached, reset
                    state <= init;

            end case;
        end if;
    end process;

    -- Additional outputs to aid simulation
    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;
```