Darnell Chen
Lab 7 Report
ECE 2031 L02
05 March 2025

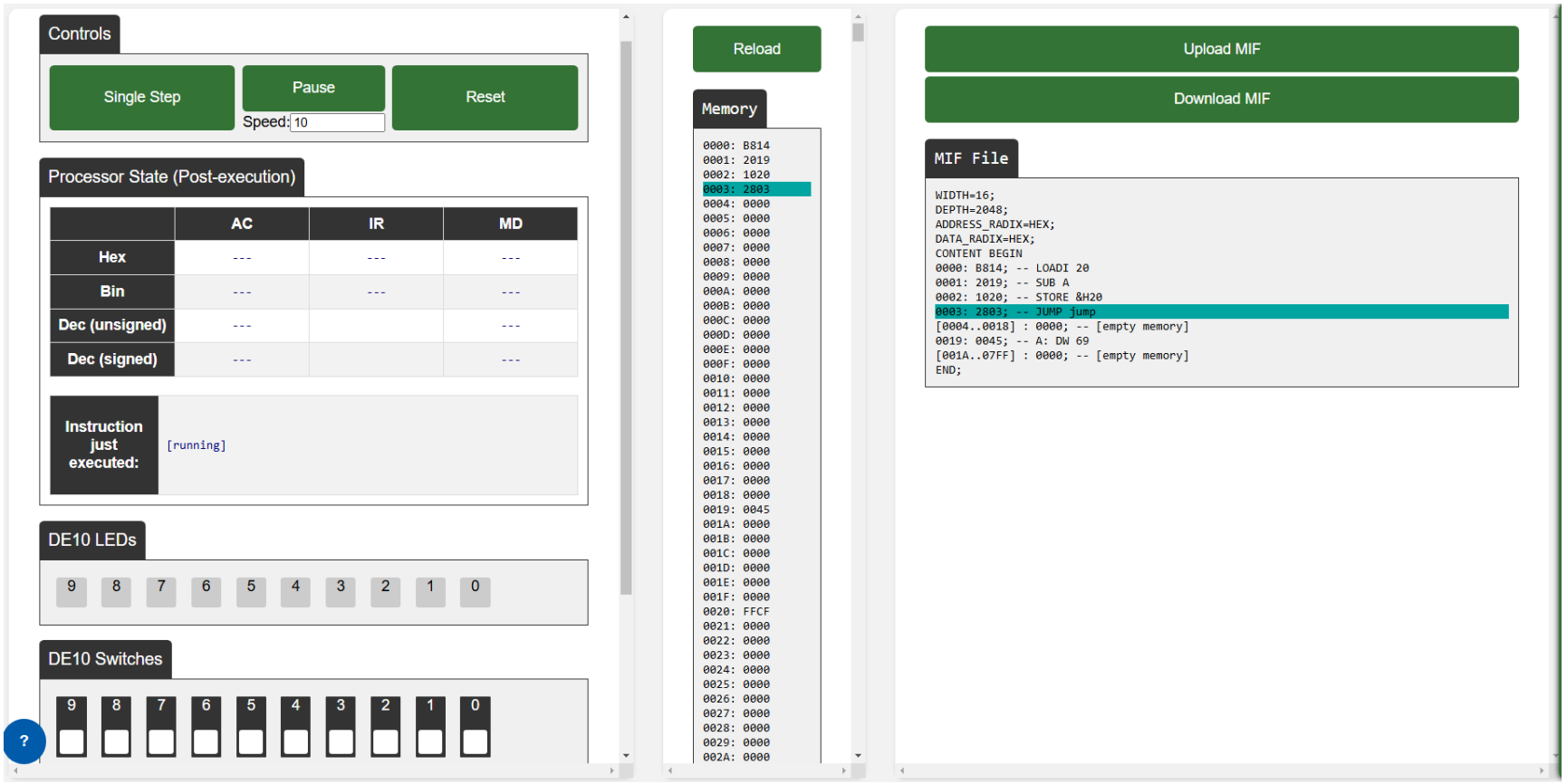

Figure 1. An online assembler that's running in an infinite loop after subtracting the value 69 from 20.

```
-- SimpleDemo.asm
-- Code that loads the value 20 and subtracts it from the
contents at memory address 0x1F before storing the result.
-- Darnell Chen
-- ECE2031 L02
-- 02/25/2025

ORG 0
     LOADI     20
     SUB       A
     STORE     &H20
jump:
     JUMP jump


ORG 25
     A:   DW   69
```

Figure 2. An assembly program code which loads the value 20 into the AC before subtracting the value stored at memory address 0x1F and storing the result in memory address 0x20.

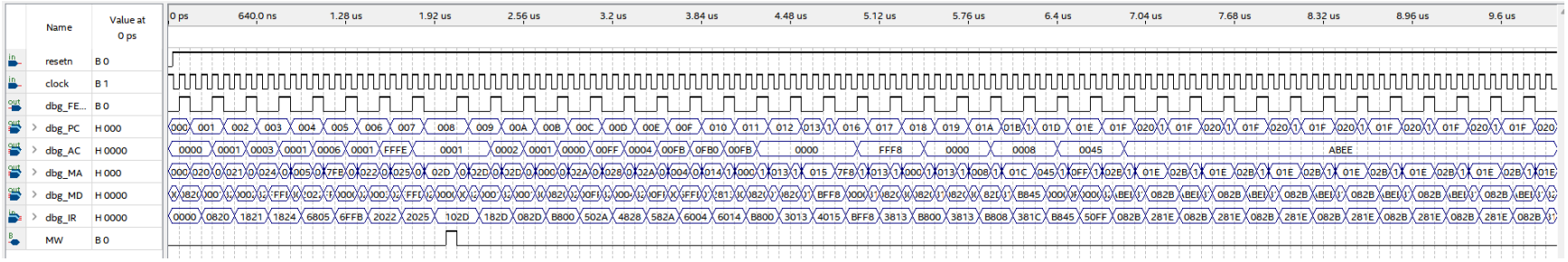

Figure 3. A Functional waveform simulation which runs a series of instructions. Here, we fixed our SUB and JPOS instructions correctly, which can be seen when PC is 005 and by the fact that our JPOS results in AC eventually becoming 0Xabee.

APPENDIX A

VHDL WHICH IMPLEMENTS SCOMP ASSEMBLY INSTRUCTIONS

```
-- SCOMP.VHD (VHDL)
-- Basic implementation of SCOMP's assembly instructions
-- Darnell Chen
-- ECE2031 L02
-- 02/25/2025


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_jump, ex_jneg, ex_jzero,
          ex_return, ex_call,
          ex_and, ex_or, ex_xor, ex_shift,
          ex_in, ex_in2, ex_out, ex_out2, ex_sub, ex_jpos
     );

     -- 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",
		clock_enable_output_a => "BYPASS",
		lpm_hint => "ENABLE_RUNTIME_MOD=NO",
		intended_device_family => "CYCLONE V",
		clock_enable_input_a => "BYPASS",
		lpm_type => "altsyncram",
		operation_mode => "SINGLE_PORT",
		power_up_uninitialized => "FALSE",
		read_during_write_mode_port_a => "NEW_DATA_NO_NBE_READ",
		outdata_reg_a => "UNREGISTERED",
		outdata_aclr_a => "NONE",
		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
            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 "00101"  =>        -- jump
                                 state <= ex_jump;
                            when "00110"  =>        -- jneg
                                 state <= ex_jneg;
                            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 "00100" =>
                                          state <= ex_sub;      --
subtract
                                    when "00111" =>
                                          state <= ex_jpos;
                                    when others =>
                                          state <= fetch;     -- invalid
opcodes don't execute
                              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_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_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 =>            -- shift is
accomplished with a dedicated shifter
                        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 ex_sub =>
                              AC <= AC - mem_data;
                              state <= fetch;

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

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

                    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_IR    <= IR;
          dbg_MA    <= next_mem_addr;
          dbg_MD    <= mem_data;

end a;
```

Appendix B

Assembly which compares and returns the larger beginning and ending nibble of a value

```
-- BitComparator.asm
-- A program that compares the lowest and highest nibbles of a
value, and returns whichever is larger
-- Darnell Chen
-- ECE2031 L02
-- 02/25/2025

ORG 0
LOAD VALUE
AND LSBBITS


STORE LSB

LOAD VALUE
SHIFT -12
AND LSBBITS
STORE MSB

SUB LSB
JNEG LsbLarger
JPOS MsbLarger

LsbLarger:
LOAD LSB
STORE RESULT
Jump Finish

MsbLarger:
LOAD MSB
STORE RESULT
JUMP Finish

Finish:
JUMP Finish



ORG 200
VALUE:    DW   &HF00E
LSB: EQU 512
```

```
MSB:      EQU 516
RESULT: EQU 1024

LSBBITS: DW &H000F
```