Matthew Liu
Lab Report 8
ECE 2031 L09
13 March 2019

```
; Lab8_PRELAB.ASM implements D = (A AND B) XOR C
; Lab 8 SCOMP and I/O Disassembly Lab,
LAB8_PRELAB.asm/mif file
; Altera Memory Initialization File (MIF) for
Pre-lab
; Matthew Liu
; ECE2031 L09
; 13 March 2019

        ORG     0
Start:  CALL    CALC    ; Jump to CALC
subroutine
Here:   JUMP    START   ; loop infinitely here

        ORG     &H010
CALC:   LOAD    A       ;CALC Subroutine
        AND     B
        XOR     C
        STORE   D
        Return

        ORG     &H0030  ;Start address 0x030
A:      DW      &H00FF  ;0x030
B:      DW      &HA5A5  ;0x031
C:      DW      &H3300  ;0x032
D:      DW      &H0000  ;0x033
```

Figure 1. Lab_Prelab.asm assembly code implements the D = (A AND B) XOR C logic through the CALC subroutine (functions from SCOMP VHDL file). The code tests the validity of the results from using the Jump function to executes a subroutine and also tests other functions included in the code. The assembly file is compiled by SCASM to generate a MIF file readable by SCOMP.

```
-- Lab8_PRELAB.mif
-- Lab 8 SCOMP and I/O Disassembly Lab
-- Altera Memory Initialization File (MIF) for Pre-lab 8
-- Implements D = (A AND B) XOR C
-- Matthew Liu
-- ECE2031 L09
-- 13 March 2019

DEPTH = 1024;
WIDTH = 16;

ADDRESS_RADIX = HEX;
DATA_RADIX = HEX;

CONTENT
  BEGIN
    [000..3FF] : 0000;  -- Default to NOP

           000 : 4010;  -- Start:   CALL    CALC    ; Jump to CALC subroutine
           001 : 1400;  -- Here:    JUMP    START   ; loop infinitely here
           010 : 0430;  -- CALC:    LOAD    A       ;CALC Subroutine
           011 : 2431;  --      AND     B
           012 : 2C32;  --      XOR     C
           013 : 0833;  --      STORE   D
           014 : 4400;  --      Return
           030 : 00FF;  -- A:       DW      &H00FF  ;0x030
           031 : A5A5;  -- B:       DW      &HA5A5  ;0x031
           032 : 3300;  -- C:       DW      &H3300  ;0x032
           033 : 0000;  -- D:       DW      &H0000  ;0x033
  END;
```

Figure 2. Lab_Prelab.mif is generated by assembly code implementing the D = (A AND B) XOR C logic through the CALC subroutine (functions from SCOMP VHDL file). The code tests the validity of the results from using the Jump function to execute a subroutine and also tests other functions included in the code. The assembly file was compiled by SCASM and readable by SCOMP.

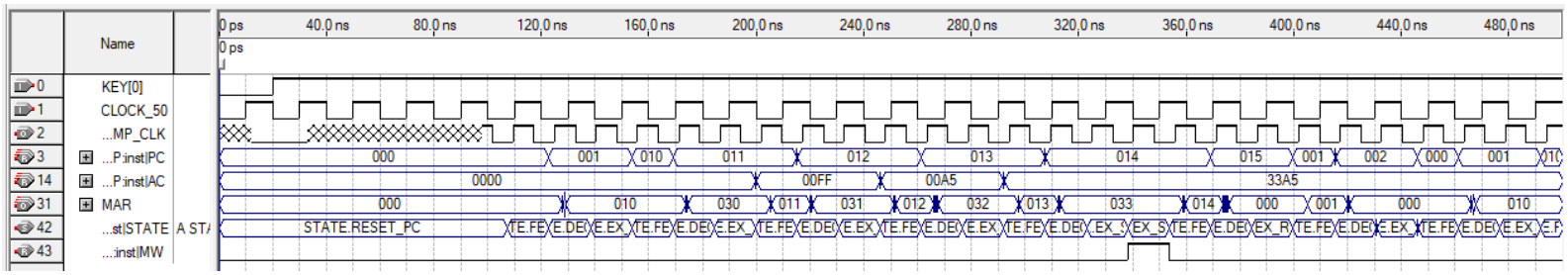


Figure 3. Quartus simulation of Lab8__PRELAB.ASM implements the CALC subroutine on the Simple Computer, D = (A and B) and C, and utilizes states/functions defined in VHDL file. Key[0] is the active-low reset, and SCOMP clock is generated from Clock_50. The expected value of MAR:&H010 at the end returns back to beginning of the CALC subroutine).

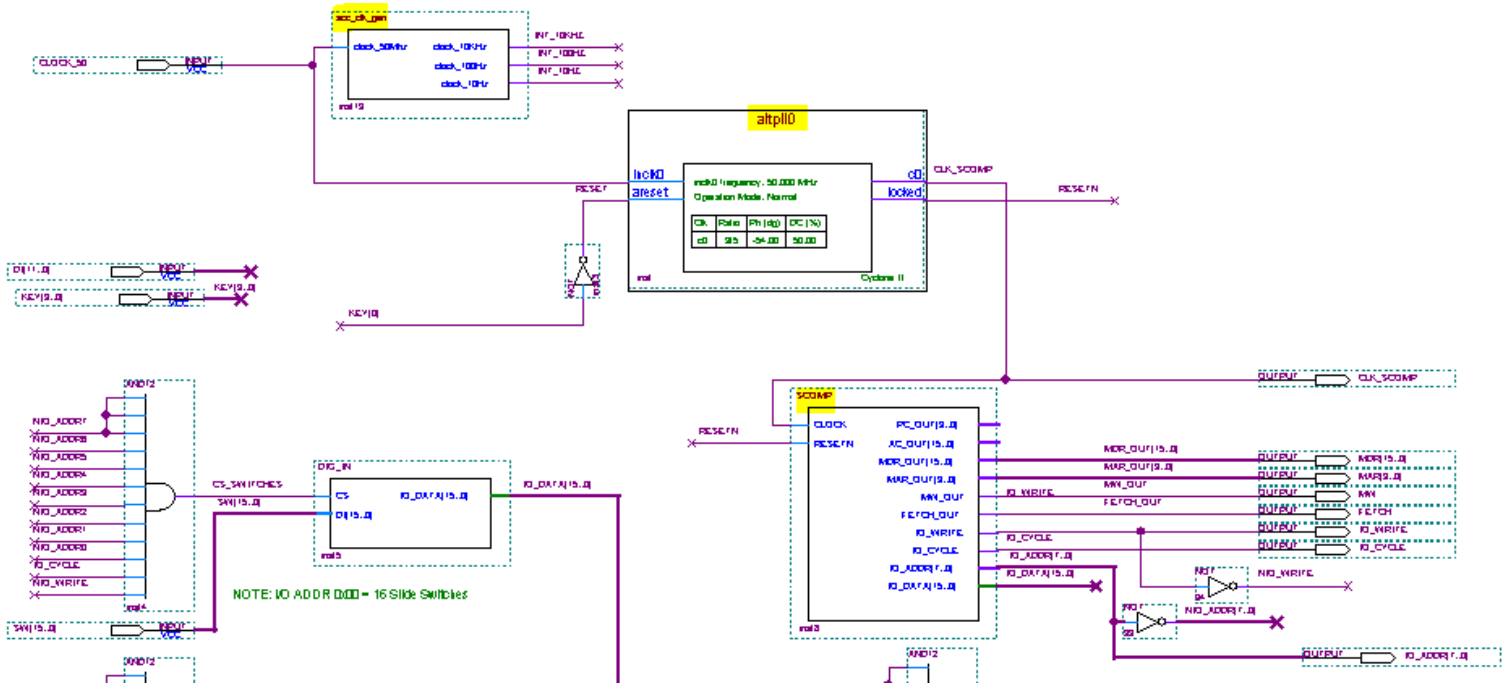


Figure 4. TOP_SCOMP.BDF with connected clock signal, and cropped to area with the PLL and SCOMP. The 50 MHz clock generates the SCOMP clock through the PLL which is the input into SCOMP. When compiled with the SCOMP.VHD file and programmed to DE2 board, the schematic generates signals to the board based on pin assignments. The BDF schematic "controls" switches, timer, seven segment displays, and LEDs.

```
; Lab8_Step7.ASM
; Lab 8 SCOMP and I/O Disassembly Lab, LAB8_Step7.asm file
; Implements Flow Chart 8.18
; Output: Write INDATA (left shifted 1 bit) value to LEDS and 7 segment
displays
; Matthew Liu
; ECE2031 L09
; 13 March 2019

            ORG     &H000   ;Begin Program

; flow chart fig 8.18
; display the value for slide switches for two seconds,
; and then shift up one bit every two seconds
Start:      IN  SWITCHES        ; 1. Read Slide Switches
            STORE   INDATA      ; 2. Store as "INDATA"

Loop:       LOAD    INDATA
            OUT     LEDS        ; 1. Write INDATA value to the LEDs
            OUT     SEVENSEG    ; 2. Write INDATA to 7 segment displays
            SHIFT   1           ; 3. Left Shift INDATA one bit
            STORE   INDATA          ;Replace the value
            OUT     TIMER           ;Reset Timer (to 0)

Loop2:      IN      TIMER       ; Read timer
            ADDI    -20         ; if two seconds elapsed then(Timer - 20)>= 0

            JPOS    Loop        ; Jump back to Loop if two second elapsed
(positive or neg)
            JZERO   Loop
            JNEG    Loop2       ; return back to timer if less than 2 seconds



            ORG     &H020           ;store data starting at x020
INDATA:     DW      &H0000
SWITCHES:   EQU     &H00        ; EQU Statements from fig 8.17
LEDS:       EQU     &H01
TIMER:      EQU     &H02
SEVENSEG:   EQU     &H04
```

Figure 5. Lab8_Step7 assembly file implements figure 8.18 of the lab manual. It displays left shifted data on LEDs and the seven segment display based on inputs from slide switches and the timer. The purpose of the code is to provide test code for newly written I/O functions in SCOMP.vhd by checking outputs from the DE2 board. The assembly file is compiled by SCASM to generate a mif file for SCOMP.

```
-- Lab8_Step7.mif
-- Lab 8 SCOMP and I/O Disassembly Lab, LAB8_Step7.mif file
-- Altera Memory Initialization File (MIF)
-- Implements Flow Chart 8.18
-- Write INDATA (left shifted 1 bit) value to LEDS and 7 segment displays
-- Matthew Liu
-- ECE2031 L09
-- 13 March 2019

DEPTH = 1024;
WIDTH = 16;

ADDRESS_RADIX = HEX;
DATA_RADIX = HEX;

CONTENT
  BEGIN
    [000..3FF] : 0000;  -- Default to NOP

           000 : 4800;  -- Start:   IN      SWITCHES ; 1. Read Slide Switches
           001 : 0820;  --          STORE   INDATA   ; 2. Store as "INDATA"
           002 : 0420;  -- Loop:    LOAD    INDATA
           003 : 4C01;  --          OUT     LEDS
                  ; 1. Write INDATA value to the LEDs

           004 : 4C04;  --          OUT     SEVENSEG
                  ; 2. Write INDATA to 7 segment displays

           005 : 3001;  --          SHIFT   1
                  ; 3. Left Shift INDATA one bit

           006 : 0820;  --          STORE   INDATA   ;Replace the value
           007 : 4C02;  --          OUT     TIMER    ;Reset Timer (to 0)
           008 : 4802;  -- Loop2:   IN      TIMER    ; Read timer

           009 : 37EC;  --          ADDI    -20
                  ; if two seconds elapsed then (Timer - 20) >= 0

           00A : 1C02;  --          JPOS    Loop
                  ; Jump back to Loop if two second elapsed (positive or neg)

           00B : 2002;  --          JZERO   Loop
           00C : 1808;  --          JNEG    Loop2
                  ; return back to timer if less than 2 seconds

           020 : 0000;  -- INDATA:      DW      &H0000
  END;
```

Figure 6. Lab8_Step7 mif file implements figure 8.18 of the lab manual. It displays left shifted data on LEDs and the seven segment display based on inputs from slide switches and the timer. The purpose of the code is to provide test code for newly written I/O functions in SCOMP.vhd by checking outputs from the DE2 board. The mif file is generated by SCASM for inputting into SCOMP init_file.

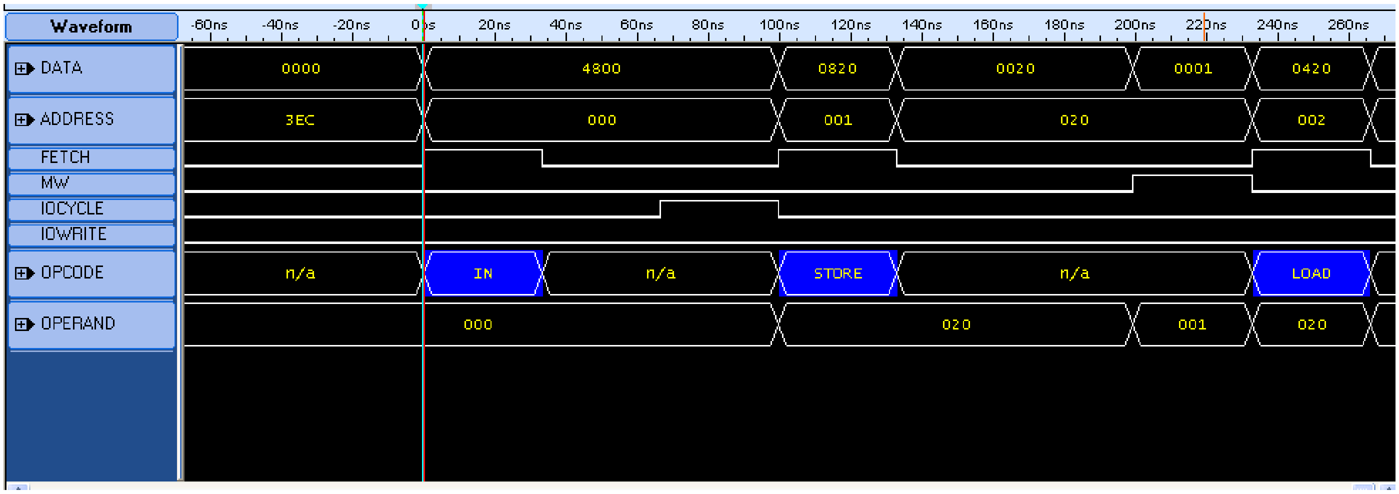
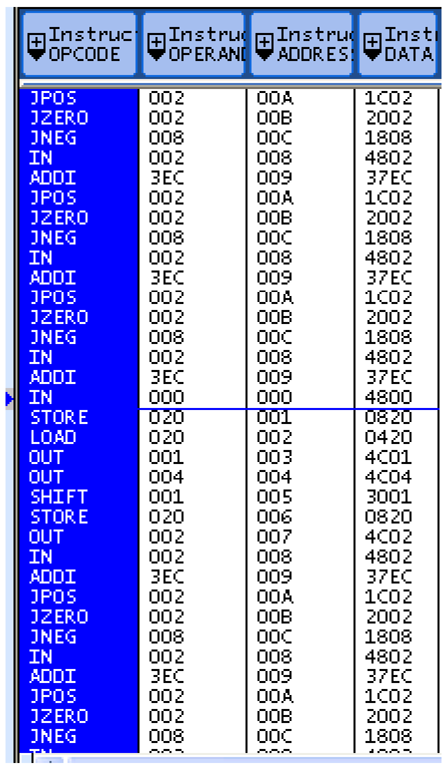


Figure 7. Logic analyzer produced disassembled waveform implementing the simple computer project and assembly program based on flow chart in figure 8.18. A useful disassembly technique used in obtaining this waveform is external clocking from DE2 board. Note that the logic analyzer trigger point occurs when DATA(MDR) is 0x4800, and the expected results (as shown by white vertical line) is that this would occur during the fetching of the instruction IN.



Instruc OPCODE	Instruc OPERAN	Instruc ADDRES	Inst DATA
JPOS	002	00A	1C02
JZERO	002	00B	2002
JNEG	008	00C	1808
IN	002	008	4802
ADDI	3EC	009	37EC
JPOS	002	00A	1C02
JZERO	002	00B	2002
JNEG	008	00C	1808
IN	002	008	4802
ADDI	3EC	009	37EC
JPOS	002	00A	1C02
JZERO	002	00B	2002
JNEG	008	00C	1808
IN	002	008	4802
ADDI	3EC	009	37EC
IN	000	000	4800
STORE	020	001	0820
LOAD	020	002	0420
OUT	001	003	4C01
OUT	004	004	4C04
SHIFT	001	005	3001
STORE	020	006	0820
OUT	002	007	4C02
IN	002	008	4802
ADDI	3EC	009	37EC
JPOS	002	00A	1C02
JZERO	002	00B	2002
JNEG	008	00C	1808
IN	002	008	4802
ADDI	3EC	009	37EC
JPOS	002	00A	1C02
JZERO	002	00B	2002
JNEG	008	00C	1808

Figure 8. Logic analyzer produced disassembled list (filtered with Fetch = 1) implementing the simple computer project and assembly program based on flow chart in figure 8.18. A useful disassembly technique used in obtaining this waveform is external clocking from DE2 board. Note that the logic analyzer trigger point occurs when DATA(MDR) is 0x4800, and the expected results (as shown by blue horizontal line) is that this would occur during the fetching of the instruction IN.

APPENDIX A
LAB 8 SCOMP VHDL IMPLEMENTATION

```
-- scomp.vhd (final version)
-- Lab 8 SCOMP/ I/O Disassembly Lab
-- VHDL File for implementing functions and hardware of SCOMP
-- Includes: CALL, RETURN, IN, OUT, SHIFT functions needed in Lab 8
-- Matthew Liu
-- ECE2031 L09
-- 7 March 2019

LIBRARY IEEE;
LIBRARY ALTERA_MF;
LIBRARY LPM;

USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_ARITH.ALL;
USE IEEE.STD_LOGIC_UNSIGNED.ALL;
USE ALTERA_MF.ALTERA_MF_COMPONENTS.ALL;
USE LPM.LPM_COMPONENTS.ALL;

ENTITY SCOMP IS
  PORT(
    CLOCK    : IN    STD_LOGIC;
    RESETN   : IN    STD_LOGIC;
    PC_OUT   : OUT   STD_LOGIC_VECTOR( 9 DOWNTO 0);
    AC_OUT   : OUT   STD_LOGIC_VECTOR(15 DOWNTO 0);
    MDR_OUT  : OUT   STD_LOGIC_VECTOR(15 DOWNTO 0);
    MAR_OUT  : OUT   STD_LOGIC_VECTOR( 9 DOWNTO 0);
    MW_OUT   : OUT   STD_LOGIC;
    FETCH_OUT: OUT   STD_LOGIC;
    IO_WRITE : OUT   STD_LOGIC;
    IO_CYCLE : OUT   STD_LOGIC;
    IO_ADDR  : OUT   STD_LOGIC_VECTOR( 7 DOWNTO 0);
    IO_DATA  : INOUT STD_LOGIC_VECTOR(15 DOWNTO 0)
  );
END SCOMP;


ARCHITECTURE a OF SCOMP IS
  TYPE STATE_TYPE IS (
    RESET_PC,
    FETCH,
    DECODE,
    EX_LOAD,
    EX_STORE,
    EX_STORE2,
    EX_ADD,
    EX_JUMP,
    EX_AND,
    EX_SUB,
    EX_JNEG,
    EX_JPOS,
    EX_JZERO,
    EX_OR,
    EX_XOR,
    EX_ADDI,
    EX_SHIFT,   -- Lab 8
    EX_CALL,    -- Lab 8
    EX_RETURN,  -- Lab 8
```

```
    EX_IN,      -- Lab 8
    EX_OUT,     -- Lab 8
    EX_OUT2
  );

  SIGNAL STATE    : STATE_TYPE;
  SIGNAL AC       : STD_LOGIC_VECTOR(15 DOWNTO 0);
  SIGNAL IR       : STD_LOGIC_VECTOR(15 DOWNTO 0);
  SIGNAL MDR      : STD_LOGIC_VECTOR(15 DOWNTO 0);
  SIGNAL PC       : STD_LOGIC_VECTOR(9 DOWNTO 0);
  SIGNAL MEM_ADDR : STD_LOGIC_VECTOR(9 DOWNTO 0);
  SIGNAL MW       : STD_LOGIC;
  SIGNAL AC_SHIFTED : STD_LOGIC_VECTOR(15 DOWNTO 0);
  SIGNAL PC_STACK   : STD_LOGIC_VECTOR(9 DOWNTO 0);
  SIGNAL IO_WRITE_INT: STD_LOGIC;


  BEGIN
    -- Use LPM_BUSTRI fucntion to drive I/O bus (Lab 8 step)
    IO_BUS: LPM_BUSTRI
    GENERIC MAP(
        lpm_width   => 16
    )
    PORT MAP(
        data        => AC,
        enabledt    => IO_WRITE_INT,
        tridata     => IO_DATA
    );
    -- Use LPM_CLSHIFT component for Logical Shift function (Prelab 8 step)
    SHIFTER: LPM_CLSHIFT
    GENERIC MAP (
        lpm_width       => 16,
        lpm_widthdist   => 4,
        lpm_shifttype   => "LOGICAL"
    )
    PORT MAP(
        data        => AC,
        distance    => IR (3 DOWNTO 0),
        direction   => IR(4),
        result      => AC_SHIFTED
    );
    -- Use altsyncram component for unified program and data memory
    MEMORY : altsyncram
    GENERIC MAP (
    intended_device_family => "Cyclone",
    width_a          => 16,
    widthad_a        => 10,
    numwords_a       => 1024,
    operation_mode   => "SINGLE_PORT",
    outdata_reg_a    => "UNREGISTERED",
    indata_aclr_a    => "NONE",
    wrcontrol_aclr_a => "NONE",
    address_aclr_a   => "NONE",
    outdata_aclr_a   => "NONE",
    init_file        => "lab8step6.mif", --example.mif, TEST_CODE.mif,
LAB8_PRELAB.mif
    lpm_hint         => "ENABLE_RUNTIME_MOD=NO",
```

```
    lpm_type         => "altsyncram"
    -- implementing shift
    --lpm_width     => 16,
    --lpm_widthdist => 4,
    --lpm_shifttype => "LOGICAL"
    )
    PORT MAP (
    wren_a    => MW,
    clock0    => NOT(CLOCK),
    address_a => MEM_ADDR,
    data_a    => AC,
    q_a       => MDR
    -- implementing shift
    --data      => AC,
    --distance  => IR(3 DOWNTO 0),
    --direction     => IR(4),
    --result        => AC_SHIFTED
    );

    PC_OUT   <= PC;
    AC_OUT   <= AC;
    MDR_OUT  <= MDR;
    MAR_OUT  <= MEM_ADDR;
    IO_ADDR  <= IR(7 DOWNTO 0);
    MW_OUT   <= MW;                          -- produce values for signal
MW_OUT

    WITH STATE SELECT
      FETCH_OUT <= '1' WHEN FETCH,           -- produce value for signal
FETCH_OUT
                   '0' WHEN OTHERS;          -- FETCH_OUT is high when in
Fetch State, which is prior to decode state

    WITH STATE SELECT
      MEM_ADDR <= PC WHEN FETCH,
                  IR(9 DOWNTO 0) WHEN OTHERS;
    --IO_WRITE <= '0'; newly implemented for I/O
    WITH STATE SELECT
      IO_WRITE <= '1' WHEN EX_OUT2,
                  '0' WHEN OTHERS;
    -- IO_CYCLE <= '0'; newly implemented for I/O
    WITH STATE SELECT
      IO_CYCLE <= '1' WHEN EX_OUT2,
                  '1' WHEN EX_IN,
                  '0' WHEN OTHERS;

    PROCESS (CLOCK, RESETN)
      BEGIN
        IF (RESETN = '0') THEN           -- Active low, asynchronous reset
          STATE <= RESET_PC;
        ELSIF (RISING_EDGE(CLOCK)) THEN
          CASE STATE IS
            WHEN RESET_PC =>
              MW        <= '0';          -- Clear memory write flag
              PC        <= "0000000000"; -- Reset PC to the beginning of
memory, address 0x000
              AC        <= x"0000";      -- Clear AC register
```

```
            STATE     <= FETCH;

          WHEN FETCH =>
            MW    <= '0';             -- Clear memory write flag
            IR    <= MDR;             -- Latch instruction into the IR
            PC    <= PC + 1;          -- Increment PC to next instruction
address
            IO_WRITE_INT <= '0';
            STATE <= DECODE;

          WHEN DECODE =>
            CASE IR(15 downto 10) IS
              WHEN "000000" =>        -- No Operation (NOP)
                STATE <= FETCH;
              WHEN "000001" =>        -- LOAD
                STATE <= EX_LOAD;
              WHEN "000010" =>        -- STORE
                STATE <= EX_STORE;
              WHEN "000011" =>        -- ADD
                STATE <= EX_ADD;
              WHEN "000101" =>        -- JUMP
                STATE <= EX_JUMP;
              WHEN "001001" =>        -- AND
                STATE <= EX_AND;
              WHEN "000100" =>         -- SUB
                STATE <= EX_SUB;
              WHEN "000110" =>         -- JNEG
                STATE <= EX_JNEG;
              WHEN "000111" =>         -- JPOS
                STATE <= EX_JPOS;
              WHEN "001000" =>         -- JZERO
                STATE <= EX_JZERO;
              WHEN "001010" =>         -- OR
                STATE <= EX_OR;
              WHEN "001011" =>         -- XOR
                STATE <= EX_XOR;
              WHEN "001101" =>         -- ADDI
                STATE <= EX_ADDI;
              WHEN "001100" =>         -- SHIFT
                STATE <= EX_SHIFT;
              WHEN "010000" =>         -- CALL
                STATE <= EX_CALL;
              WHEN "010001" =>         -- RETURN
                STATE <= EX_RETURN;
                                              -- during IN/OUT operation
IO_Cycle goes High
              WHEN "010010" =>         -- IN
                STATE <= EX_IN;
                 --IO_CYCLE <= '1';
              WHEN "010011" =>         -- OUT
                IO_WRITE_INT <= '1';
                STATE <= EX_OUT;
                --IO_CYCLE <= '1';
              WHEN OTHERS =>
                STATE <= FETCH;       -- Invalid opcodes default to NOP
            END CASE;
```

```
            WHEN EX_LOAD =>
              AC    <= MDR;             -- Latch data from MDR (memory
contents) to AC
              STATE <= FETCH;

            WHEN EX_STORE =>
              MW    <= '1';             -- Raise MW to write AC to MEM
              STATE <= EX_STORE2;

            WHEN EX_STORE2 =>
              MW    <= '0';             -- Drop MW to end write cycle
              STATE <= FETCH;

            WHEN EX_ADD =>
              AC    <= AC + MDR;
              STATE <= FETCH;

            WHEN EX_JUMP =>
              PC    <= IR(9 DOWNTO 0);
              STATE <= FETCH;

            WHEN EX_AND =>
              AC    <= AC AND MDR;
              STATE <= FETCH;

            WHEN EX_SUB =>
              AC    <= AC - MDR;
              STATE <= FETCH;

            WHEN EX_JNEG =>
              if (AC(15) = '1') then -- AND AC < 1?
                PC  <= IR(9 DOWNTO 0);
              end if;
              STATE <= FETCH;

            WHEN EX_JPOS =>
              if (AC(15) = '0' AND AC /= x"0000") then
                PC  <= IR(9 DOWNTO 0);
              end if;
              STATE <= FETCH;

            WHEN EX_JZERO =>
              if AC = x"0000" then
                PC  <= IR(9 DOWNTO 0);
              end if;
              STATE <= FETCH;

            WHEN EX_OR =>
              AC    <= AC OR MDR;
              STATE <= FETCH;

            WHEN EX_XOR =>
              AC    <= AC XOR MDR;
              STATE <= FETCH;

            WHEN EX_ADDI=>
```

```
              AC <= AC + (IR(9) & IR(9) & IR(9) & IR(9) & IR(9) & IR(9) &
IR(9 DOWNTO 0));
              STATE <= FETCH;

            WHEN EX_SHIFT=>
              AC <= AC_SHIFTED;
              STATE <= FETCH;

            WHEN EX_CALL =>
              PC_STACK <= PC;
              PC <= IR(9 DOWNTO 0);
              STATE <= FETCH;

            WHEN EX_RETURN =>
              PC <= PC_STACK;
              STATE <= FETCH;

            WHEN EX_IN =>
              AC <= IO_DATA;             -- IO_WRITE remains low, data latched
as IO_CYCLE goes low
              STATE <= FETCH;

            WHEN EX_OUT =>
               STATE <= EX_OUT2;

            WHEN EX_OUT2 =>
              IO_WRITE_INT <= '0';
              STATE <= FETCH;

            WHEN OTHERS =>
              STATE <= FETCH;          -- If an invalid state is reached,
return to FETCH
          END CASE;
        END IF;
      END PROCESS;
  END a;
```