Paul Kafka

10/21/13

Embedded System Design

Homework 5

VHDL

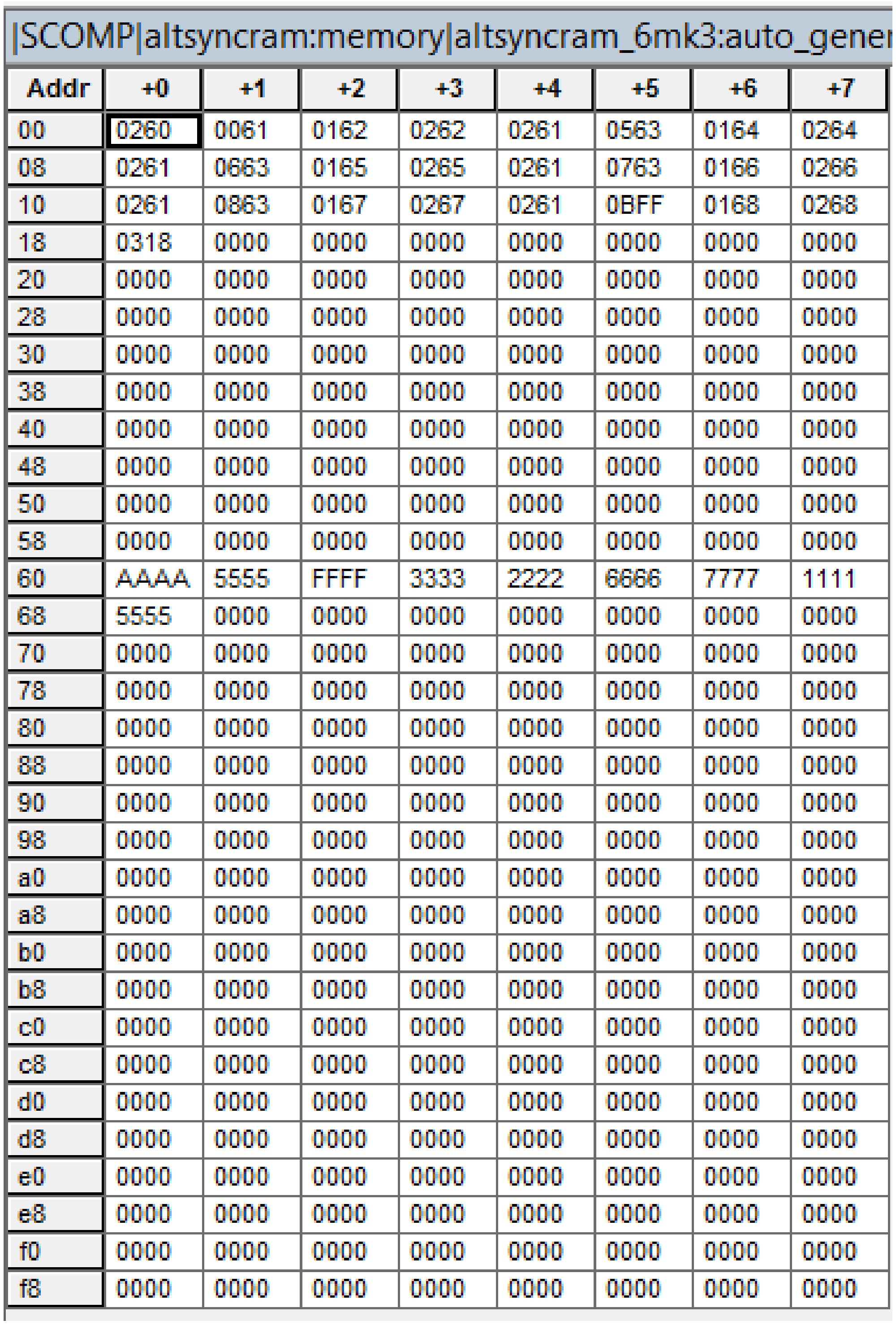
```
LIBRARY IEEE;
USE  IEEE.STD_LOGIC_1164.ALL;
USE  IEEE.STD_LOGIC_ARITH.ALL;
USE  IEEE.STD_LOGIC_UNSIGNED.ALL;
LIBRARY altera_mf;
USE altera_mf.altera_mf_components.all;

ENTITY SCOMP IS
PORT(   clock, reset                   : IN STD_LOGIC;
        program_counter_out           : OUT STD_LOGIC_VECTOR( 7 DOWNTO 0 );
        register_AC_out               : OUT STD_LOGIC_VECTOR(15 DOWNTO 0 );
        memory_data_register_out      : OUT STD_LOGIC_VECTOR(15 DOWNTO 0 );
        memory_address_register_out : OUT STD_LOGIC_VECTOR(7 DOWNTO 0 ));
END SCOMP;

ARCHITECTURE a OF scomp IS
TYPE STATE_TYPE IS (
                     ------------ADDED-------------------------------------------------
                    execute_subt, execute_xor, execute_or, execute_and, execute_addi,
                     ----------------------------------------------------------------
-
                     reset_pc, fetch, decode, execute_add, execute_load,
                     execute_store, execute_store3, execute_store2, execute_jump );

SIGNAL state: STATE_TYPE;
SIGNAL instruction_register, memory_data_register   : STD_LOGIC_VECTOR(15 DOWNTO 0 );
SIGNAL register_AC                 : STD_LOGIC_VECTOR(15 DOWNTO 0 );
SIGNAL program_counter             : STD_LOGIC_VECTOR( 7 DOWNTO 0 );
SIGNAL memory_address_register  : STD_LOGIC_VECTOR( 7 DOWNTO 0 );
SIGNAL memory_write                : STD_LOGIC;

BEGIN
            -- Use Altsyncram function for computer's memory (256 16-bit words)
  memory: altsyncram
      GENERIC MAP (
        operation_mode => "SINGLE_PORT",
        width_a => 16,
        widthad_a => 8,
        lpm_type => "altsyncram",
        outdata_reg_a => "UNREGISTERED",
            -- Reads in mif file for initial program and data values
        init_file => "program.mif",
        intended_device_family => "Cyclone")
```

```vhdl
    PORT MAP (  wren_a => memory_write, clock0 => clock,
                address_a => memory_address_register, data_a => Register_AC,
                q_a => memory_data_register );
            -- Output major signals for simulation
        program_counter_out         <= program_counter;
        register_AC_out             <= register_AC;
        memory_data_register_out    <= memory_data_register;
        memory_address_register_out <= memory_address_register;

PROCESS ( CLOCK, RESET )
    BEGIN
    IF reset = '1' THEN
        state <= reset_pc;
    ELSIF clock'EVENT AND clock = '1' THEN
        CASE state IS
                -- reset the computer, need to clear some registers
        WHEN reset_pc =>
            program_counter         <= "00000000";
--          memory_address_register <= "00000000";
            register_AC             <= "0000000000000000";
            memory_write            <= '0';
            state                   <= fetch;
                -- Fetch instruction from memory and add 1 to PC
        WHEN fetch =>
            instruction_register    <= memory_data_register;
            program_counter         <= program_counter + 1;
            memory_write            <= '0';
            state                   <= decode;
                -- Decode instruction and send out address of any data operands
        WHEN decode =>
            CASE instruction_register( 15 DOWNTO 8 ) IS
                WHEN X"00" =>
                    state   <= execute_add;
                WHEN X"01" =>
                    state   <= execute_store;
                WHEN X"02" =>
                    state   <= execute_load;
                WHEN X"03" =>
                    state   <= execute_jump;
                ----------Added-------------------------
                WHEN X"05" =>
                    state   <= execute_subt;
                WHEN X"06" =>
                    state   <= execute_xor;
                WHEN X"07" =>
                    state   <= execute_or;
                WHEN X"08" =>
                    state   <= execute_and;
                WHEN X"0B" =>
                    state   <= execute_addi;
                ----------------------------------------
                WHEN OTHERS =>
                    state   <= fetch;
            END CASE;
        ----------------ADDED---------------------------------------------------
        WHEN execute_subt=>
            register_ac                 <= register_ac - memory_data_register;
            state                       <= fetch;
        WHEN execute_xor=>
            register_ac                 <= register_ac XOR memory_data_register;
            state                       <= fetch;
        WHEN execute_or=>
            register_ac                 <= register_ac OR memory_data_register;
            state                       <= fetch;
        WHEN execute_and=>
            register_ac                 <= register_ac AND memory_data_register;
            state                       <= fetch;
        WHEN execute_addi=>
            if memory_data_register =X"FF" THEN
                register_ac                 <= register_ac - 1;
            else
                register_ac                 <= register_ac + memory_data_register;
            state                       <= fetch;
            END IF;
        ---------------------------------------------------------------------

        -- Execute the ADD instruction
        WHEN execute_add =>
            register_ac                 <= register_ac + memory_data_register;
            state                       <= fetch;
                    -- Execute the STORE instruction
                    -- (needs three clock cycles for memory write)
        WHEN execute_store =>
                    -- write register_AC to memory
            memory_write                <= '1';
            state                       <= execute_store2;
                    -- This state ensures that the memory address is
                    --  valid until after memory_write goes inactive
        WHEN execute_store2 =>
            memory_write                <= '0';
            state                       <= execute_store3;
        WHEN execute_store3 =>
            state                       <= fetch;
                    -- Execute the LOAD instruction
        WHEN execute_load =>
            register_ac                 <= memory_data_register;
            state <= fetch;
                    -- Execute the JUMP instruction
        WHEN execute_jump =>
            program_counter             <= instruction_register( 7 DOWNTO 0 );
            state                       <= fetch;
        WHEN OTHERS =>
            state <= fetch;
        END CASE;
    END IF;
   END PROCESS;

-- memory address register is stored inside synchronous memory unit
-- need to send it's outputs based on current state
   WITH state SELECT
        memory_address_register <= "00000000"                  WHEN reset_pc,
                             program_counter             WHEN fetch,
                             instruction_register(7 DOWNTO 0)   WHEN decode,
                             program_counter             WHEN execute_add,
                             ---------------ADDED-------------------------------
                             program_counter             WHEN execute_subt,
                             program_counter             WHEN execute_xor,
                             program_counter             WHEN execute_or,
                             program_counter             WHEN execute_and,
                             program_counter             WHEN execute_addi,
                             ---------------------------------------------------
                             instruction_register(7 DOWNTO 0)   WHEN execute_store,
                             instruction_register(7 DOWNTO 0)   WHEN execute_store2,
                             program_counter             WHEN execute_store3,
                             program_counter             WHEN execute_load,
                             instruction_register(7 DOWNTO 0)   WHEN execute_jump;

END a;
```

MIF File

```
DEPTH = 256;    % Memory depth and width are required   %
WIDTH = 16;     % Enter a decimal number    %

ADDRESS_RADIX = HEX;    % Address and value radixes are optional    %
DATA_RADIX = HEX;       % Enter BIN, DEC, HEX, or OCT; unless   %
                        % otherwise specified, radixes = HEX    %
-- Specify values for addresses, which can be single address or range
CONTENT
    BEGIN
[00..FF]    : 0000; % Range--Every address from 00 to FF = 0000 (Default) %
---------Test for the ADD instruction ------------------------------------
00 :0260; % LOAD AC with MEM(60)=B=AAAA  %
01 :0061; % ADD MEM(61)=C=5555 to AC %
02 :0162; % STORE AC in MEM(62)=A %
03 :0262; % LOAD AC with MEM(62) check for new value of FFFF %
---------Test for the SUBI instruction -----------------------------------
04 :0261; % Load AC with MEM(61)=C %
05 :0563; % Test C-D = 2222. 05 is minus opcode. 63 is D's Address  %
06 :0164; % STORE AC in MEM(64)=E %
07 :0264; % LOAD AC with MEM(64) check for new value of 2222 %
---------Test for the XOR instruction ------------------------------------
08 :0261; % Load AC with MEM(61)=C %
09 :0663; % Test C XOR D = 6666. 06 is XOR opcode. 63 is D's Address %
0A :0165; % STORE AC in MEM(65)=F %
0B :0265; % LOAD AC with MEM(65) check for new value of 6666 %
---------Test for the OR instruction -------------------------------------
0C :0261; % Load AC with MEM(61)=C %
0D :0763; % Test C OR D= 7777.  07 is OR opcode. 63 is D's Address %
0E :0166; % STORE AC in MEM(66)=G %
0F :0266; % LOAD AC with MEM(66) check for new value of 7777 %
---------Test for the AND instruction ------------------------------------
10 :0261; % Load AC with MEM(61)=C %
11 :0863; % Test C AND D = 1111.  08 is AND opcode. 63 is D's Address %
12 :0167; % STORE AC in MEM(67)=H %
13 :0267; % LOAD AC with MEM(67) check for new value of 1111 %
---------Test for the ANDI instruction -----------------------------------
14 :0261; % Load AC with MEM(61)=C %
15 :0BFF; % Test C ANDi FF = 5554.  0B is ANDI opcode. FF is the integer %
16 :0168; % STORE AC in MEM(68)=I %
17 :0268; % LOAD AC with MEM(68) check for new value of 5554 %
-------------------------------------------------------------------------
18  :0318;  % JUMP to same line (loop forever) 03 is jump opcode. 18 is same line %
-------------------------------------------------------------------------
60  :AAAA;  % Data Value of B %
61  :5555;  % Data Value of C %
62  :0000;  % Data Value of A - should be FFFF after running program %
63  :3333;  % Data Value of D %
64  :0000;  % Data value of E - should be 2222 after running program %
65  :0000;  % Data value of F - should be 6666 after running program %
66  :0000;  % Data value of G - should be 7777 after running program %
67  :0000;  % Data value of H - should be 1111 after running program %
68  :0000;  % Data value of I - should be 5554 after running program %
END ;
```

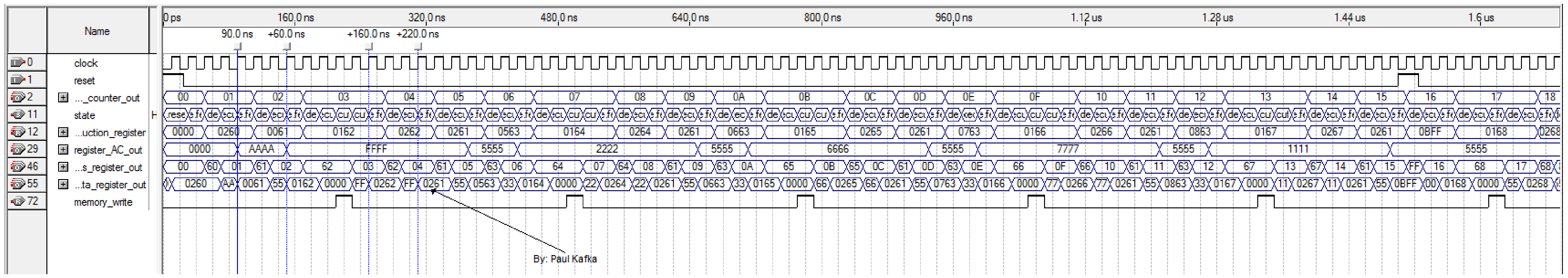
Memory After Simulation



|SCOMP|altsyncram:memory|altsyncram_6mk3:auto_gener

Addr	+0	+1	+2	+3	+4	+5	+6	+7
00	0260	0061	0162	0262	0261	0563	0164	0264
08	0261	0663	0165	0265	0261	0763	0166	0266
10	0261	0863	0167	0267	0261	0BFF	0168	0268
18	0318	0000	0000	0000	0000	0000	0000	0000
20	0000	0000	0000	0000	0000	0000	0000	0000
28	0000	0000	0000	0000	0000	0000	0000	0000
30	0000	0000	0000	0000	0000	0000	0000	0000
38	0000	0000	0000	0000	0000	0000	0000	0000
40	0000	0000	0000	0000	0000	0000	0000	0000
48	0000	0000	0000	0000	0000	0000	0000	0000
50	0000	0000	0000	0000	0000	0000	0000	0000
58	0000	0000	0000	0000	0000	0000	0000	0000
60	AAAA	5555	FFFF	3333	2222	6666	7777	1111
68	5555	0000	0000	0000	0000	0000	0000	0000
70	0000	0000	0000	0000	0000	0000	0000	0000
78	0000	0000	0000	0000	0000	0000	0000	0000
80	0000	0000	0000	0000	0000	0000	0000	0000
88	0000	0000	0000	0000	0000	0000	0000	0000
90	0000	0000	0000	0000	0000	0000	0000	0000
98	0000	0000	0000	0000	0000	0000	0000	0000
a0	0000	0000	0000	0000	0000	0000	0000	0000
a8	0000	0000	0000	0000	0000	0000	0000	0000
b0	0000	0000	0000	0000	0000	0000	0000	0000
b8	0000	0000	0000	0000	0000	0000	0000	0000
c0	0000	0000	0000	0000	0000	0000	0000	0000
c8	0000	0000	0000	0000	0000	0000	0000	0000
d0	0000	0000	0000	0000	0000	0000	0000	0000
d8	0000	0000	0000	0000	0000	0000	0000	0000
e0	0000	0000	0000	0000	0000	0000	0000	0000
e8	0000	0000	0000	0000	0000	0000	0000	0000
f0	0000	0000	0000	0000	0000	0000	0000	0000
f8	0000	0000	0000	0000	0000	0000	0000	0000

Simulation


Name
0 ps
160.0 ns
320.0 ns
480.0 ns
640.0 ns
800.0 ns
960.0 ns
1.12 us
1.28 us
1.44 us
1.6 us
90.0 ns
+60.0 ns
+160.0 ns
+220.0 ns
0 clock
1 reset
2 ..._counter_out
11 state
12 ...uction_register
29 register_AC_out
46 ...s_register_out
55 ...ta_register_out
72 memory_write
By: Paul Kafka