Jiacheng Zhang

Lab 7 Report

ECE 2031 L07

04 Mar 2022

表格

低可信度描述已自动生成

**Figure 1.** Assembly code implementing the functionality of adding nine when the stored value is even or dividing the stored value by two and adding one if it is odd, then storing the result to memory location 0x100.

图片包含 图形用户界面

描述已自动生成

**Figure 2.** A simulation waveform demonstrating the assembly code (sequence.asm) that in an infinite loop, if the stored value is even, add 9, and if the value is odd, divide it by 2 and add 1.

文本

中度可信度描述已自动生成

**Figure 3.** Assembly code implementing a “bouncing” animation on the LEDs on the DE10-Standard. When the pattern hits the left or right side of the LEDs and reverses direction, set the bit that hit the side to zero.

图形用户界面, 文本, 应用程序

中度可信度描述已自动生成

**Figure 4.** Assembly code implementing a game on the DE10-Standard that matches the state of the LEDs on the switches.

APPENDIX A   
VHDL IMPLEMENTING A SIMPLE COMPUTER

-- SCOMP, the Simple Computer.

-- This VHDL defines a simple 16-bit processor that is easy to

-- understand and modify.

-- Jiacheng Zhang

-- ECE2031 L07

-- 03/04/2022

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\_jpos, ex\_jzero,

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 => "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\_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

-- opcode is top 5 bits of instruction

case mem\_data(15 downto 11) is

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" => -- subtraction

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;