Rudra Goel

Lab 07 Report

ECE 2031 L10

10 October 2024

A screenshot of a computer

Description automatically generated

**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.’

A screenshot of a graph

Description automatically generated

**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.

A screenshot of a computer

Description automatically generated

**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;