CPU Design Project Part 5 - Final CPU

Joe Driscoll

April 19, 2019

Contents

1 CPU Verification with Test Program

To test the final CPU, a large test program was created to test various use cases for the CPU, as well as test nearly every instruction. The test program created heavily depended on memory loads and stores, adds, immediate loads, logical operations, branch and links, subroutines, and more. An assembler was also created in the Python programming language to assist in translating the eventual, large test program.

The correct execution of many of these were tested manually by looking at the signals in the simulation window, but the overall correct execution of the program as a whole implies that each of the constituent instructions work (since each instruction has a purpose in the test program).

The task of the test program is to constantly update register 1 with the value of ASCII characters in a string hard-coded into the program. This could lazily be done by loading immediate values into the register, but the test program instead first stores all capital letter ASCII values into the first 26 contiguous memory locations. Then, the program does a permanent loop, where each character is placed into the register by calling a subroutine, which places a character based on the argument passed to the subroutine in register 0. These two levels of complexity, as well as various added complexities (performing usually simple operations with many instructions) made the program test nearly every instruction. While the program could easily be changed to print out a different sequence of letters, the string chosen was "SO THATS A THING". The string being printed out in one iteration of the permanent/main loop is shown in Figure 1, which is simply the contents of register 1 seen by the register file debugging port required by the specifications.

Figure 2 shows the instruction being executed at any given times, as well as source and destination registers for each instruction. This shows the process in which a character is loaded from memory, which is done by placing a number 0-25 into register 0 and calling bl to jump to the subroutine that loads the character to register 1. The subroutine offsets a register holding zero (r14) by r0 to select the desired character from memory. This made selecting a capital ASCII character easier to choose, since only the cardinal value of the letter need be known, rather than the ASCII code for the character itself. For example, the main loop can load the letter "C" into r1 by loading r0 with 2, rather than manually loading r1 with the ASCII value for "C".

Figure 3 shows the beginning of the program, where the ASCII codes for the characters A-Z are loaded into the first twenty-five data memory locations. This is done in a loop, and the index variable contained in r1 is compared to the value zero to figure out when the memory initialization is done. In the code, this is done within the "memory_init_loop" label. The branch seen in Figure 3 shows the structured nature of the initialization (using loops rather than loading each explicitly).

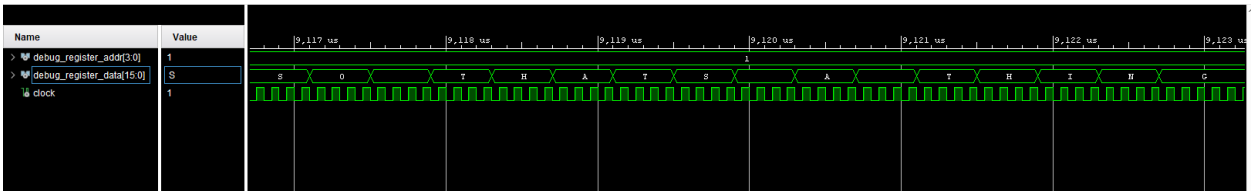

Figure 1. Signal view showing that the string "SO THATS A THING" is correctly printed out.

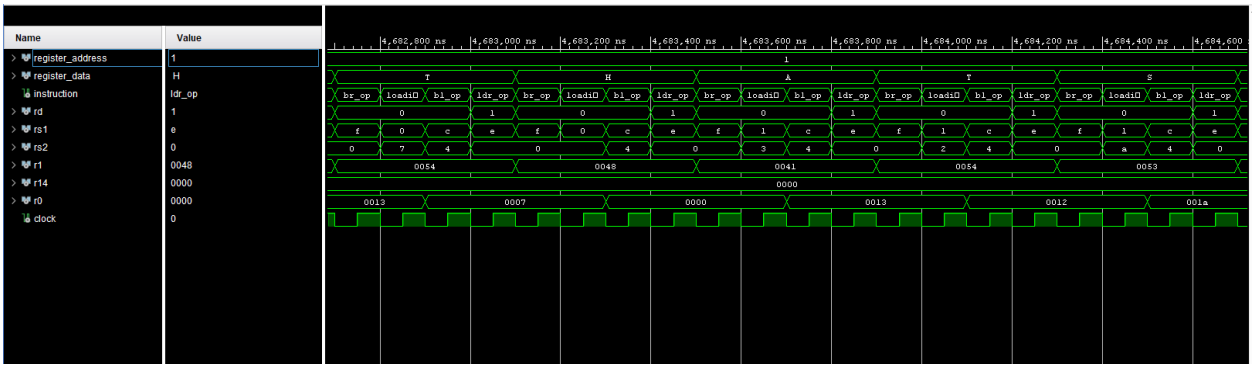

Figure 2. Signal view showing the character printing process in the main loop, with the instruction executing being shown to demonstrate the use of a subroutine to load characters into register 1.

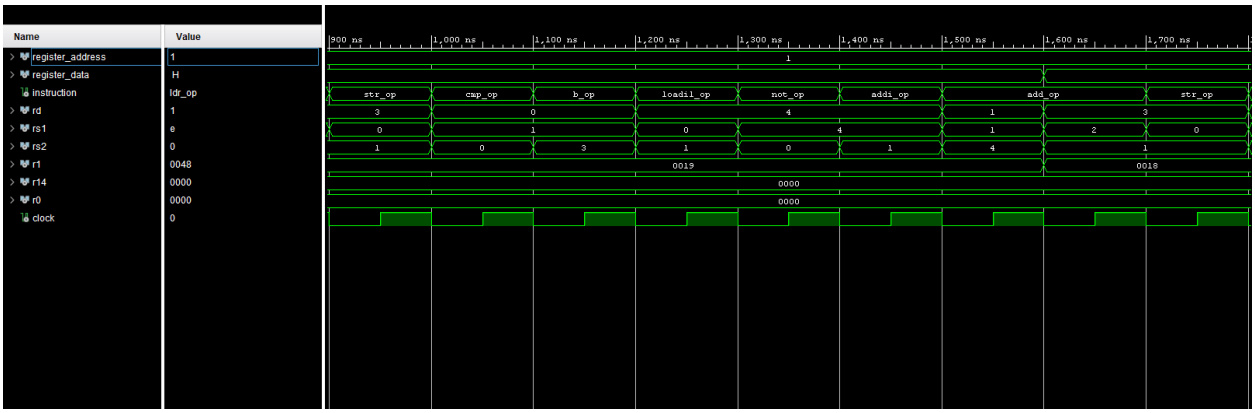

Figure 3. Signal view showing the ASCII characters being written to memory at the beginning of the program.

A Instruction Memory HDL

```
-- File: instruction_memory.vhd
--
-- Implements an instruction memory for the CPU.


library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;


-- Memory interface.
entity instruction_memory is

    port    (read_address  : in  std_logic_vector(9 downto 0);
             data_out      : out std_logic_vector(15 downto 0));

end instruction_memory;


-- Register implementation.
architecture behavioral of instruction_memory is

    -- Types for internal representation of memory.
    subtype word_t               is std_logic_vector(15 downto 0);
    type    instruction_memory_t is array(0 to 2**10 - 1) of word_t;

    constant main_program : instruction_memory_t := ("0000000000001011",
                                                     "0001000110101100",
                                                     "0001000110001110",
                                                     "0010010000011011",
                                                     "0100000000011011",
                                                     "0100010000000110",
                                                     "0100010000011101",
                                                     "0001000101000000",
                                                     "0011001000010000",
                                                     "0011000000010010",
                                                     "0000000100000111",
                                                     "0000000100111001",
                                                     "0011001000001011",
                                                     "0100000110101011",
                                                     "0011010000000010",
                                                     "1110000000001011",
                                                     "0000000100101011",
                                                     "0000110001001010",
                                                     "0000000011101011",
                                                     "0000110001001010",
                                                     "0000000110101011",
                                                     "0000110001001010",
                                                     "0000000100111011",
                                                     "0000110001001010",
                                                     "0000000001111011",
                                                     "0000110001001010",
                                                     "0000000000001011",
                                                     "0000110001001010",
                                                     "0000000100111011",
                                                     "0000110001001010",
                                                     "0000000100101011",
                                                     "0000110001001010",
                                                     "0000000110101011",
                                                     "0000110001001010",
                                                     "0000000000001011",
                                                     "0000110001001010",
                                                     "0000000110101011",
                                                     "0000110001001010",
                                                     "0000000100111011",
                                                     "0000110001001010",
                                                     "0000000001111011",
                                                     "0000110001001010",
                                                     "0000000010001011",
                                                     "0000110001001010",
                                                     "0000000011011011",
                                                     "0000110001001010",
                                                     "0000000001101011",
                                                     "0000110001001010",
                                                     "0000001111001001",
                                                     "0001111000000011",
                                                     "0000111100001000",
                                                     others => (others => '0'));
```

```
    -- Instruction memory as a signal.
    signal internal_storage : instruction_memory_t := main_program;

begin

    data_out <= internal_storage(to_integer(unsigned(read_address)));

end behavioral;
```

B Data Memory HDL

```
-- File: data_memory.vhd
--
-- Implements a data memory for the CPU.


library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;


-- Memory interface.
entity data_memory is

    port    (address        : in  std_logic_vector(9 downto 0);
             data_out       : out std_logic_vector(15 downto 0);
             write_enable   : in  std_logic;
             write_data     : in  std_logic_vector(15 downto 0));

end data_memory;


-- Register implementation.
architecture behavioral of data_memory is

    -- Types for internal representation of memory.
    subtype word_t        is std_logic_vector(15 downto 0);
    type    data_memory_t is array(((2**10) - 1) downto 0) of word_t;

    -- Data memory as a signal.
    signal data_memory_storage : data_memory_t := (others => (others => '0'));

begin

    process(write_enable, write_data)
    begin
        if (write_enable = '1') then
            data_memory_storage(to_integer(unsigned(address))) <= write_data;
        end if;
    end process;

    data_out <= data_memory_storage(to_integer(unsigned(address)));

end behavioral;
```

C Top-level CPU HDL

```
-- File: cpu.vhd
--
-- Top-level HDL implementation of the CPU.


library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.types.all;


library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.types.all;
```

```
library work;
use work.all;
use work.instruction_memory;


entity cpu is

    port (debug_register_address    : in  std_logic_vector(3 downto 0);
          debug_register_data       : out std_logic_vector(15 downto 0);
          clock                     : in  std_logic);

end cpu;


architecture behavioral of cpu is

    -- Memory buses.
    signal memory_input_bus     : memory_input_bus_t;
    signal memory_output_bus    : memory_output_bus_t;

begin

    -- Datapath instantiation.
    datapath : entity work.datapath

        port map (debug_register_address    => debug_register_address,
                  debug_register_data       => debug_register_data,
                  memory_input_bus          => memory_input_bus,
                  memory_output_bus         => memory_output_bus,
                  clock                     => clock);


    -- Instruction memory instantiation.
    instruction_memory : entity work.instruction_memory

        port map (read_address => memory_output_bus.instruction_address_bus,
                  data_out     => memory_input_bus.instruction_read_bus);


    -- Data memory instantiation.
    data_memory : entity work.data_memory

        port map (address  => memory_output_bus.data_address_bus,
                  data_out      => memory_input_bus.data_read_bus,
                  write_enable  => memory_output_bus.data_write_enable,
                  write_data    => memory_output_bus.data_write_bus);

end behavioral;
```

D Test Assembly Program

```
loadil  r0, 0
loadiu  r1, 26
lsr     r1, r1, 8
loadil  r2, 0x41
memory_init_loop:
loadil  r4, 1
not     r4, r4
addi    r4, r4, 1
add     r1, r1, r4
add     r3, r2, r1
str     r3, r0, r1
cmp     r1, r0
bgt     memory_init_loop
loadil  r3, 0x20
loadil  r4, 26
str     r3, r4, r0
main_loop:
loadil  r14, 0
loadil  r0, 18
bl      print_function
loadil  r0, 14
bl      print_function
loadil  r0, 26
bl      print_function
loadil  r0, 19
```

```
bl      print_function
loadil  r0, 7
bl      print_function
loadil  r0, 0
bl      print_function
loadil  r0, 19
bl      print_function
loadil  r0, 18
bl      print_function
loadil  r0, 26
bl      print_function
loadil  r0, 0
bl      print_function
loadil  r0, 26
bl      print_function
loadil  r0, 19
bl      print_function
loadil  r0, 7
bl      print_function
loadil  r0, 8
bl      print_function
loadil  r0, 13
bl      print_function
loadil  r0, 6
bl      print_function
b       main_loop
print_function:
ldr     r1, r14, r0
br      r15
```

E Assembler

```
import ply.lex as lex
import ply.yacc as yacc
import re


symbol_table = {}


tokens = ("INSTRUCTION",
          "LABEL")


def t_LABEL(t):
    r'.+:'

    symbol_table[t.value[:-1]] = str(t.lexer.lineno - 1)
    t.lexer.lineno = t.lexer.lineno - 1

    t.value = {'label': t.value[:-1]}


def t_INSTRUCTION(t):
    r'.+'

    # Extract tokens from instruction line.
    instruction_tokens = re.sub('[,]*[ ]+[,]*', ',', t.value).split(',')

    # Initialize value to dict.
    if instruction_tokens[0][0] == 'b':

        if len(instruction_tokens[0]) < 3:

            t.value = {'op'     : instruction_tokens[0],
                       'cond'   : 'none'}

        elif len(instruction_tokens[0]) == 3:

            t.value = {'op'     : instruction_tokens[0][0],
                       'cond'   : instruction_tokens[0][1:]}

        elif len(instruction_tokens[0]) == 4:

            t.value = {'op'     : instruction_tokens[0][0:2],
                       'cond'   : instruction_tokens[0][2:]}

    else:
```

```
        t.value = {'op': instruction_tokens[0]}

    # Pull out operands of instruction and put them into t.value.
    for i in range(1, len(instruction_tokens)):

        t.value['arg' + str(i - 1)] = instruction_tokens[i]

    # Add in all zeros for unused arg2 for not instruction.
    if t.value['op'] == 'not':
        t.value['arg2'] = '0'

    return t


def t_newline(t):
    r'\n+'
    t.lexer.lineno += len(t.value)


def t_error(p):
    print("Syntax error at %s" %p.value)


# Ignore commas, spaces, and tabs.
t_ignore = ', \t'


# Build lexer.
lexer = lex.lex()


# Testing.
test_program = ""
with open('program.txt', 'r') as file:
    test_program = file.read()


# Instead of using parser, directly manipulate lextokens from lexer.

# Op code mappings.
op_code_mapping = {'add'   : '0000',
                   'sub'   : '0001',
                   'str'   : '0010',
                   'ldr'   : '0011',
                   'and'   : '0100',
                   'or'    : '0101',
                   'not'   : '0110',
                   'cmp'   : '0111',
                   'br'    : '1000',
                   'b'     : '1001',
                   'bl'    : '1010',
                   'loadil': '1011',
                   'loadiu': '1100',
                   'addi'  : '1101',
                   'lsr'   : '1110',
                   'lsl'   : '1111'}

# Condition branch code mappings
conditional_mapping = {'none'   : '00',
                       'eq'     : '01',
                       'lt'     : '10',
                       'gt'     : '11'}


# Convert decimal number into n-bit binary number.
def decimal_to_binary_string(decimal, n_bits):

    output = ""

    if decimal[0:2] == '0x':
        output = ('{0:0'+ str(n_bits) + 'b}').format(int(decimal, 16))
    else:
        output = ('{0:0' + str(n_bits) + 'b}').format(int(decimal))

    return output


# Convert register into number.
def register_number_to_binary(register):
```

```
    output = ""

    if register[0] == 'r':
        output = '{0:04b}'.format(int(register[1:]))
    else:
        output = '{0:04b}'.format(int(register))

    return output



def generate_machine_code(token):

    output = ""
    opcode = op_code_mapping[token.value['op']]

    # Handle branches.
    if token.value['op'][0] == 'b':

        # Convert branch condition to binary.
        condition = conditional_mapping[token.value['cond']]

        # Handle register branches and other branches differently.
        if token.value['op'][0:2] == 'br':

            # Extract register number.
            register_number = register_number_to_binary(token.value['arg0'])

            # Form output binary.
            output = '0000' + register_number + '00' + condition

        else:

            # Extract branch address.
            branch_address = decimal_to_binary_string(symbol_table[token.value['arg0']], 10)

            # Form output binary.
            output = branch_address + condition

    # Compare instructions have different first argument.
    elif token.value['op'] == 'cmp':

        # Extract source registers from token.
        rs1 = register_number_to_binary(token.value['arg0'])
        rs2 = register_number_to_binary(token.value['arg1'])

        output = '0000' + rs1 + rs2

    # Loadiu and loadil have a weirdly sized constant.
    elif token.value['op'][0:4] == 'load':

        # Extract source register from token.
        rs1 = register_number_to_binary(token.value['arg0'])

        # Extract 8-bit constant.
        constant = decimal_to_binary_string(token.value['arg1'], 8)

        # Form output binary.
        output = rs1 + constant

    else:

        # Extract source and destination registers from token.
        rd  = register_number_to_binary(token.value['arg0'])
        rs1 = register_number_to_binary(token.value['arg1'])
        rs2 = register_number_to_binary(token.value['arg2'])

        output = rd + rs1 + rs2

    # Opcode always appended to end.
    return "\"" + (output + opcode)  + "\","



# Input test program to lexer.
lexer.input(test_program)

toks = []

while True:
```

```
    tok = lexer.token()

    if not tok:
        break

    toks.append(tok)


for token in toks:
    print(generate_machine_code(token))
```