FINAL ASSIGNMENT

OWN MICROCONTROLLER

Design based on:

ESP32 ULP Co-Processor

Saxion University of Applied Sciences

Smart Embedded Systems Minor

Embedded Systems – Spring 2020 / 2021

Student:

Luciano Regis Orlandini – 460952

Tutors:

Mr. Christiaen Slot

Mr. Ghayoor Gillani

Contents

Preface

This document describes the entire work developed by the student Luciano Regis Orlandini on the final assignment of the Embedded Systems course, part of the Smart Embedded Systems minor at Saxion University of Applied Sciences.

It begins with an introduction and the definition of requirements set out by the student in Moscow format. Once these requirements were approved by the tutors the work began on the development of the software required to run the microcontroller.

The document includes a detailed description of how the programme works and includes code in assembly-style language for easier understanding.

Finally, it includes a personal reflection detailing the entire thought process taken by the student to conclude this assignment and highlight key learns and challenges faced during the process.

1 Introduction

1.1 Assignment

This assignment required the student to create his own microcontroller design, based on existing instruction sets encountered on common designs such as the AVR or the PIC controllers.

1.2 Methodology

Due to current COVID-19 measures, the student will use software RAM and will write the code in VHDL. This code will be combined into a testbench and run with ModelSim. It is not planned to be synthesised and programmed into actual hardware.

From ModelSim the student will be able to showcase the microcontroller running through its instructions in order to display the wave showing the correct functioning of the VHDL code.

2 Controller Design

The student has based his own microcontroller on the ESP32 Ultra-Low-Power (ULP) co-processor which is found in an Espressif ESP32-WROOM chip. The co-processor is a finite state machine which can only be programmed in Assembly language. Normally during compile time, this Assembly code gets translated into binary and included in the main programme as a binary blob. The student will modify this design slightly to make the work more manageable.

2.1 Original Design

Below follows a description of some of the co-processor configurations as set out by Espressif.

The ULP co-processor (Espressif) has 4x 16-bit general purpose registers and an 8-bit counter register used for loops.

It can also access 8k bytes of memory which is addressed in 32-bit word units. All instructions are 32-bits wide.

2.2 Student Design

The student will maintain some of the configurations mentioned above and will tweak / add some as per the following table in Moscow format.

One key change is that on top of the 4 general purpose registers (R1..4), there will be a 5th register (R0) which will serve as an accumulator by being the default destination of any ALU instructions.

MoSCoW

MUST HAVE	SHOULD HAVE
1x 16-bit accumulator register	Separate memory modules for code and data
4x 16-bit general purpose registers	Up to 16x 16-bit general purpose registers
10x Assembly instructions (Refer to list below)	Include a zero flag
Access to 256 words of memory	1x 8-bit stage count register for loops
Output and write enable pins	Additional Instructions (Refer to list below)
22-bit instructions (Refer to table below)	
Include a carry flag	
COULD HAVE	**WILL NOT HAVE**
Access to more memory	Peripheral connectivity (I2C, ADC, SPI…)
More instructions	Will not be synthesised
Include a kill-switch	Will not be programmed into hardware

Instructions Set

MUST HAVE

Instruction	Condition
SET *K*	N/A
ADD *Rx, Rx*	N/A
SUB *Rx, Rx*	N/A
AND *Rx, Rx*	N/A
OR *Rx, Rx*	N/A
NOT *Rx*	N/A
MOVE *Rsrc, Rdst*	N/A
ST *Mem, Rsrc*	N/A
LD *Mem, Rdst*	N/A
JUMP *Cond, Addr*	ZF*, CF*

ZF – Zero Flag; CF = Carry Flag;

SHOULD HAVE

Instruction	Condition
JUMPS *Cond, K Addr*	LE*, GE*
STAGE_INC *K*	N/A
STAGE_DEC *K*	N/A
STAGE_RST	N/A

LE = Lesser or equal than; GE = Greater or equal than;

COULD HAVE

Instruction	Condition
HALT	N/A

3 Controller Architecture

3.1 Instruction Set

This programme can currently execute 14 different instructions + 'fetch'. These are defined on the table below.

Instruction	Condition	Code	Function
SET *K*	N/A	0000 00	Set R0 with 16-bit constant.
ADD *Rx, Rx*	N/A	0001 00	Add two operands. Store in R0.
SUB *Rx, Rx*	N/A	0010 00	Subtract two operands. Store in R0.
AND *Rx, Rx*	N/A	0011 00	Logical AND of two operands. Store in R0.
OR *Rx, Rx*	N/A	0100 00	Logical OR of two operands. Store in R0.
NOT *Rx*	N/A	0101 00	Logical NOT of an operand. Store in R0.
MOVE *Rsrc, Rdst*	N/A	0110 00	Move value from Rsrc to Rdst.
ST *Mem, Rsrc*	N/A	0111 00	Store data to specified memory location.
LD *Mem, Rdst*	N/A	1000 00	Load data from specified memory location.
JUMP *Cond, Addr*	ZF*, CF*	1001 01 1001 10	Jump to specified location. Condition Optional.
JUMPS *Cond, K Addr*	LE*, GE*	1010 01 1010 10	Jump based on stage count register + condition.
STAGE_INC *K*	N/A	1011 00	Increment stage count register.
STAGE_DEC *K*	N/A	1100 00	Decrement stage count register.
STAGE_RST	N/A	1101 00	Reset stage count register.
FETCH	N/A	*1111*	Fetch next instruction. Runs automatically between instructions.

3.2 Instruction Format

All instructions are 22 bits wide. Details on how an instruction is constructed to follow. Refer to the table below for a visual representation.

21 - 18	17 - 16	15-8	7 – 4	3 - 0
Instruction	Condition	K High Byte	K Low Byte Upper Nibble	K Low Byte Lower Nibble
OpCode		Memory Location	Rsrc / Rx	Rdst / Rx

- **Bits 21 – 18**

These bits contain the 4-bit wide instruction. With this current configuration there can be a maximum of 15 instructions as instance '1111' is reserved for 'fetch'.

- **Bits 17 – 16**

These bits allow the possibility of a condition to be attached to the instruction. A maximum of 3 conditions can be defined as variation '00' is reserved for 'no condition'. These 2 bits together with the 4 MSBs of the instruction form the complete OpCode.

- **Bits 15 – 8**

These bits will serve different purposes depending on the opcode. For JUMP, JUMPS, ST and LD these bits will form the 8-bit memory address. For SET, these bits form the high byte of the 16-bit value which is possible to set. They have no use for the other instructions.

- **Bits 7 – 4**

These bits will serve different purposes depending on the opcode. For ADD, SUB, AND, OR and NOT these bits for the register index of the first operand. For MOVE and ST these bits for the register index of the source register. For SET these bits form the low byte, upper nibble of the 16-bit value which is possible to set. For JUMPS these bits form the upper nibble of the 8-bit stage count target value.

- **Bits 3 – 0**

These bits will serve different purposes depending on the opcode. For ADD, SUB, AND, OR and NOT these bits form the register index of the second operand. For MOVE and LD these bits for the register index of the destination register. For SET these bits form the low byte, lower nibble of the 16-bit value which is possible to set. For JUMPS these bits form the lower nibble of the 8-bit stage count target value.

3.3 Memory Format

The memory is stored in 22-bit wide words preceded by the program counter which is currently only able to access 256 (8-bits) words. The virtual memory architecture is built to support access up to 65536 (16-bits) words. Refer to attached file 'sram.vhd' for the memory's source code.

4 Controller Execution

The controllers source code will be broken down in this chapter to provide an overview of its functionality. The programme has been written in VHDL and can be found in the attachments as 'ulp.vhd'.

4.1 Set Up

The programme will initially load the libraries required for execution. Then during the entity declaration, the I/O will be defined. The 'data' port is 22-bits wide as it will hold the running instructions.

The behaviour for the previously declared entity is then defined, including the type definition of a 2D array which holds all general purpose registers, as well as the internal signals used during execution.

The registers and its indexes are declared as variables so their state will change instantly. This is useful to set the zero flag on the same cycle as the ALU operation that resulted in a '0', so the following instruction will already be able to make use of the zero flag's condition if required.

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

entity ulp is
   port (
      data:    inout  std_logic_vector(21 downto 0);
      address: out    std_logic_vector(7 downto 0);
      oe:      out    std_logic;
      we:      out    std_logic;
      rst:     in     std_logic;
      clk:     in     std_logic);
end;

architecture behaviour of ulp is
   type registers is array(4 downto 0) of std_logic_vector(16 downto 0);

   shared variable r:   registers;
   signal zero:         std_logic;
   signal st_cnt:       unsigned(7 downto 0);
   signal addreg:       std_logic_vector(7 downto 0);
   signal pc:           std_logic_vector(7 downto 0);
   signal state:        std_logic_vector(3 downto 0);
   shared variable i, j:integer range 0 to 15;
```

4.2 Process Begin

The only defined process is triggered by a change in the state of the running clock or reset port.

If reset is clear, it will set all signals and variables to their initial state. This state is mostly '0' except for 'state' which is set to '1111' or 'fetch'. This ensures that the first action of the controller is to fetch the first instruction in memory.

If reset is set, the clock is in its rising edge, and 'state' equals 'fetch', it will increase the program counter by one, so that at the next cycle it will read the following instruction from memory, and will fill the register index variables, regardless if they will be used for the upcoming instruction.

If 'state' is not 'fetch', it will then update the address signal with the program counter so that the programme moves forward upon the next 'fetch' state.

```
begin
   process(clk, rst)
   begin

      if (rst = '0') then
         r(4 downto 1)    := (others => (others => '0'));
         r(0)             := "ZZZZZZZZZZZZZZZZ";
         zero             <= '0';
         st_cnt           <= (others => '0');
         addreg           <= (others => '0');
         pc               <= (others => '0');
         state            <= "1111";

      elsif rising_edge(clk) then
         if (state = "1111") then
            pc <= addreg + 1;
            i := to_integer(unsigned(data(7 downto 4)));
            j := to_integer(unsigned(data(3 downto 0)));
         else
            addreg    <= pc;
         end if;
```

4.3 Fetch, Decode, Execute Cycle

The programme will then check the state of 'state' and run as per the current instruction. It's in this part that the registers and 'stage count' are updated as required.

Afterwards, if 'state' is anything but 'fetch', it will set it as 'fetch' ahead of the next cycle so the programme does not stop. It then also checks if any ALU operation resulted in '0', in order to correctly set the 'zero flag'

```
case state is
    -- Fetching...
    when "1111" => null;
    -- SET k
    when "0000" => r(0) := "0" & data(15 downto 0);
    -- ADD Rx, Rx
    when "0001" => r(0) := r(i) + r(j);
    -- SUB Rx, Rx
    when "0010" => r(0) := r(i) - r(j);
    -- AND Rx, Rx
    when "0011" => r(0) := r(i) AND r(j);
    -- OR Rx, Rx
    when "0100" => r(0) := r(i) OR r(j);
    -- NOT Rx
    when "0101" => r(0)(15 downto 0) := NOT r(i)(15 downto 0);
    -- MOVE Rsrc, Rdst
    when "0110" => r(j) := r(i);
    -- ST Mem, Rsrc
    when "0111" => null;
    -- LD Mem, Rdst
    when "1000" => r(j) := data(16 downto 0);
    -- JUMP Cond, Addr
    when "1001" => zero <= '0';
    -- JUMPS Cond, Addr
    when "1010" => null;
    -- STAGE_INC k
    when "1011" => st_cnt <= st_cnt + (unsigned(data(7 downto 0)));
    -- STAGE_DEC k
    when "1100" => st_cnt <= st_cnt - (unsigned(data(7 downto 0)));
    -- STAGE_RST
    when "1101" => st_cnt <= "00000000";
    when others => null;
end case;

-- Return to fetch state and set/clear zero flag
if (state /= "1111") then
    state <= "1111";
    if (r(0) = "0000000000000000") then
          zero <= '1';
    else
          zero <= '0';
    end if;
```

If ‘state’ is either JUMP or JUMPS, the programme will then update the address and program counter signals after checking for any conditions so it will resume execution from the specified memory location.

If 'state' is either ST or LD it will only update the address signal so the programme will store or load data to/from memory at the specified address.

```
    -- JUMP
    elsif (data(21 downto 18) = "1001") then
       if (data(17 downto 16) = "00") then
          state <= "1001";
          addreg <= data(15 downto 8);
          pc <= data(15 downto 8);
       elsif (data(17 downto 16) = "01") then
          if (zero = '1') then
             state <= "1111";
             addreg <= data(15 downto 8);
             pc <= data(15 downto 8);
          elsif (zero = '0') then
             state <= "1001";
          end if;
       elsif (data(17 downto 16) = "10") then
          if (r(0)(16) = '1') then
             state <= "1111";
             addreg <= data(15 downto 8);
             pc <= data(15 downto 8);
             r(0)(16) := '0';
          else
             state <= "1001";
          end if;
       end if;

    -- JUMPS
    elsif (data(21 downto 18) = "1010") then
       if (data(17 downto 16) = "01") then
          if (st_cnt <= unsigned(data(7 downto 0))) then
             state <= "1111";
             addreg <= data(15 downto 8);
             pc <= data(15 downto 8);
          else
             state <= "1010";
          end if;
       elsif (data(17 downto 16) = "10") then
          if (st_cnt >= unsigned(data(7 downto 0))) then
             state <= "1111";
             addreg <= data(15 downto 8);
             pc <= data(15 downto 8);
          else
             state <= "1010";
          end if;
       end if;

    -- ST
    elsif (data(21 downto 18) = "0111") then
       addreg <= data(15 downto 8);
       state <= "0111";

    -- LD
    elsif (data(21 downto 18) = "1000") then
       addreg <= data(15 downto 8);
       state <= "1000";

    else
       state <= data(21 downto 18);
    end if;
end if;
end process;
```

4.4 Output

Finally, at the end of the process the output ports get updated as required per the state of the programme.

```
    -- Output
    address  <= addreg;
    data  <= "zzzzzzzzzzzzzzzzzzzzzz" when state /= "0111" else ("11110" & r(i));
    oe <= '1' when (clk='1' or state  = "0111" or rst='0' or state = "1001") else '0';
    we <= '1' when (clk='1' or state /= "0111" or rst='0') else '0';

end behaviour;
```

5 Example

In order to test the controller described above a small programme has been written to make use of all different instructions available. The complete set up and simulation will be described in this chapter.

5.1 Testbench

All executions ran and were replicated with the use of a testbench written in VHDL. This testbench was then simulated with the use of ModelSim, by integrating the entity and memory to form a CPU system.

The main part of the testbench simulates the initialisation of the controller, then it runs a 'reset' before starting the main loop which will run indefinitely.

```
process
    begin
        clk <= '0';
        reset <= '1';
        report "CPU init";
        WAIT FOR 50 ns;
        clk <= '0';
        reset <= '0';
        report "CPU reset";
        WAIT FOR 50 ns;
        clk <= '1';
        WAIT FOR 25 ns;
        reset <= '1';
        WAIT FOR 25 ns;
        report "CPU running";
        loop
            clk <= '0';
            WAIT FOR 50 ns;
            clk <= '1';
            WAIT FOR 50 ns;        -- clock.
        end loop;
        report "end";
    end process;
end;
```

The full testbench code can be found in the attachments as 'testbench.vhd'.

5.2 Code

This controller is not able to recognise code in any programming language, hence it needs to be fed directly in binary form. This can be achieved by saving a text document called 'memory.dat' into the same folder as the testbench and '.vhd' files.

Since binary is not very legible, the same code is being shown in Assembly language next to it for reference. All files are also included with the submission.

```
00 00000000000000000001
01 01100000000000000001
02 00000000000000000010
03 01100000000000000100
04 00100000000001000001
05 01100000000000000100
06 10010100001000000000
07 10010000001000000000
08 00000000000000000011
09 01100000000000000010
10 01010000000000100000
11 00010000000000000001
12 01110010000000100000
13 10000010000000000011
14 00010000000000000011
15 10011000010001000000
16 10010000000000000000
17 00000000000011111111
18 01100000000000000001
19 00000011111111110000
20 01100000000000000010
21 00110000000000010010
22 01000000000000010010
23 00000000000001100100
24 01100000000000000001
25 00000000000000001010
26 01100000000000000010
27 00100000000000010010
28 01100000000000000001
29 10110000000000011001
30 10101000100000001000110
31 10010000011011000000000
32 11000000000000111100
33 10100100000000000001
34 11010000000000000000
```

```
    .text
begin:
    SET   1             ; 00 ( R0 = 1 )
    MOVE  r0, r1        ; 01 ( R1 = R0 )
    SET   2             ; 02 ( R0 = 2 )
    MOVE  r0, r4        ; 03 ( R4 = R0 )
phase_1:
    SUB   r4, r1        ; 04 ( R0 = R4 - R1 )
    MOVE  r0, r4        ; 05 ( R4 = R0 )
    JUMP  phase_2, eq   ; 06 ( PC = 08 if (R0 = 0) )
    JUMP  phase_1       ; 07 ( PC = 04 )
phase_2:
    SET   3             ; 08 ( R0 = 2 )
    MOVE  r0, r2        ; 09 ( R2 = R0 )
    NOT   r2            ; 10 ( R0 = ~R2 )
    ADD   r0, r1        ; 11 ( R0 = R0 + R1 )
    ST    128, r2       ; 12 ( MEM[128] = R2 )
    LD    128, r3       ; 13 ( R3 = MEM[128] )
    ADD   r0, r3        ; 14 ( R0 = R0 + R3 )
    JUMP  phase_3, cf   ; 15 ( PC - 17 if (R0[16] = 1) )
    JUMP  begin         ; 16 ( PC = 0 ) *expected to be ignored
phase_3:
    SET   511           ; 17 ( R0 = 511 )
    MOVE  r0, r1        ; 18 ( R1 = R0 )
    SET   65408         ; 19 ( R0 = 65408 )
    MOVE  r0, r2        ; 20 ( R2 = R0 )
    AND   r1, r2        ; 21 ( R0 = R1 & R2 )
    OR    r1, r2        ; 22 ( R0 = R1 | R2 )
phase_4:
    SET   100           ; 23 ( R0 = 100 )
    MOVE  r0, r1        ; 24 ( R1 = R0 )
    SET   10            ; 25 ( R0 = 10 )
    MOVE  r0, r2        ; 26 ( R2 = R0 )
loop:
    SUB   r1, r2        ; 27 ( R0 = R1 - R2 )
    MOVE  r0, r1        ; 28 ( R1 = R0 )
    STAGE_INC 25        ; 29 ( ST_CNT += 25 )
    JUMPS GE, 70, end   ; 30 ( PC = 32 if (ST_CNT >= 70) )
    JUMP  loop          ; 31 ( PC = 27 )
end:
    STAGE_DEC 60        ; 32 ( ST_CNT -= 60 )
    JUMPS LE, 1, begin  ; 33 ( PC = 0 if (ST_CNT <= 1) ) *expected to be ignored
    STAGE_RST           ; 34 ( ST_CNT = 0 )
```

5.3 Wave

The simulation wave has been executed using ModelSim, but since the output is too large, it is included as an additional file.

6 Reflection

This assignment posed a challenge as it gave me the freedom to choose my own task and controller type.

I have initially planned to do something involving also a hardware input, where I would perhaps be able to send instructions serially via an Arduino or ESP32, but due to COVID-19 restrictions and the uncertainty of having access to FPGA boards, I decided to keep everything virtual.

As part of my minor, I am working on a project that uses the ESP32's ULP Co-Processor, which was the inspiration for this result I am submitting.

I have taken the simple microcontroller example available on Blackboard as a starting point, but then did an almost complete revamp in order to incorporate the additional instructions desired. That is when the first challenges arose. The current virtual memory and microcontroller were only using 8-bit memory and instructions and were implementing only limited instructions.

This set me out to have to put in a lot of work before I would be in a position to even test my programme, as after defining which instructions I wanted to implement and the size in bits that I would require for them, I had to adjust the memory source code so it would be able to handle large instructions. This required a lot of planning as I did not want to have to increase the size of my instructions mid-work, which would cause a cascade effect and have me amend the memory file again.

Due to the complexity of some of the instructions I was implementing, I also had to change the way the programme behaves, in comparison to the one I started out of. This was a challenge, as I still did not have a programme I was able to test, so had to rely on the syntax help from Quartus in addition to all the tutorials and examples I found online.

Once I got into a position of testing, seeing the simulation crash in specific points was really helpful to better understand the programme and ensure all instructions were working properly.

It was very rewarding to see the simulation run the programme in full without crashing at the end.

Overall, this assignment increased my understanding of how microcontrollers work, as in addition to the extra knowledge in VHDL I acquired, it also made me realise the path to creating a compiler, which would for example read some code written in Assembly, and decode it to create the binary instructions used by the hardware.

References

Espressif. (2021, 05 07). *ESP32 ULP Coprocessor Instruction Set*. From ESP-IDF Programming Guide: https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-guides/ulp_instruction_set.html