Study Diary - Digital Microelectronics 2

Haris Reyaz Sultanpuri

07.06.2024

Aalto University
School of Electrical
Engineering

Contents

1 VHDL description of PIC16F84A microcontroller.

1.1 Objective - Implementing an ALU using VHDL

1.1.1 20.03.2024 - WEDNESDAY

I started the exercise04 of the course. I read the exercise instructions. The goal was to design an ALU for the PIC microcontroller. The datasheet listed out all the instructions. The instructions viz DECFSZ, INCFSZ, BTFSC, BTFSS, CLRWDT, RETFIE and SLEEP were not to be implemented. Moreover, CALL, RETURN, GOTO and RETLW required Stack. As such, I decided to include them later. The suggestions from the textbook were to get familiar with *procedures, packages and functions*. As such, I began understanding these constructs of the language.

1.1.2 24.03.2024 - SUNDAY

In order to implement all the instruction of the ALU, I decided to write their functionality inside procedures. This meant there would be as many procedures as there are instructions for the PIC microcontroller. All these procedures would later be clubbed inside a package. This approach would enhance the modularity of the code.

In total there were 18 procedures to be implemented in this exercise. For this exercise, all the inputs came from a file. The inputs were Opcode, two Operands, Status Input to ALU and Bit Select to the ALU. This Bit Select input is used in BCF and BSF instructions, which clears and sets the bit pointed by this Bit Select input. To begin with, I started from the first instruction on the datasheet. Read its functionality and wrote a procedure for it. I thought it better to write the Package and Procedure simultaneously. I declared the Procedure and then defined it in the Package Body. For this exercise, we did not have to worry about whether the result was stored in Work Register or at any Address. This simplified the design. As such, instructions like ADDWF and ADDLW had the same procedure. Likewise for ANDWF and ANDLW and so on.

Another thing which I observed that, while writing these procedures some pieces of code was repetitive. Like for example, the code to check for Zero Flag, Carry out and Nibble Carry. I decided to write a function for these code snippets. At this point, I had two types of constructs, one Procedures and the other Functions. For clarity and for efficient debugging later, I decided to make two separate packages.

One for all the Procedures and another file for all the Functions.

For ADDWF and ADDLW instruction, in order to capture the carry out bit, I concatenated a zero to both the operands and put the addition value in a 9-bit variable. This 9th bit was then sent to the *msbFlag0* function to determine a carry 1 or 0. Similar idea was implemented for nibble carry. In order to check whether the result is zero or non zero, the 9 bit result was sent to the *zeroFlag2* function. This is shown in the snippet image below:

```
--1. ADDLW
 procedure ADDLW(signal W,f     : in  word8;
                 signal status_in : in  word3;
                 signal status    : out word3;
                 signal result    : out word8) is
   variable temp9 : word9;
   variable temp5 : word5;

   begin
     temp9 := STD_LOGIC_VECTOR((UNSIGNED('0' & W)) + (UNSIGNED('0' & f)));
     temp5 := STD_LOGIC_VECTOR((UNSIGNED('0' & W(3 downto 0))) +
                               (UNSIGNED('0' & f(3 downto 0))));
     status(2) <= zeroFlag2(temp9);
     status(1) <= nibbleFlag1(temp5(temp5'left));
     status(0) <= msbFlag0(temp9(temp9'left));

     result <= temp9(7 downto 0);
  end procedure ADDLW;
```

Figure 1.1: ADDWF operation

The functions zeroFlag2, nibbleFlag1 and msbFlag0 were pretty straightforward to write. The ending number in their function names indicate their bit position in the STATUS register. STATUS register is an 8-bit register in the PIC microcontroller.

The next few instructions were straightforward to write. One thing I had to be cautious about was that since the zeroFlag2 function has a 9-bit input, I had to concatenate an extra zero wherever required. I implemented eight instructions today viz: ADDLW, ANDLW, CLRF, CLRW, COMF, DECF, INCF, IORWF.

1.1.3 26.03.2024 - TUESDAY

The objective today was to finish writing all the instructions for the ALU and an entity file for ALU. I started with MOVF instruction. All the instructions were pretty straight forward to write. The BSF and BCF reuqired a bit of thinking. For BSF, the Bit Select input mentioned which bit position needed to be set to one. A variable "pos" was declared with zeroth bit as 1. Based on the Bit Select value this Zeroth bit was shifted that many locations. All this was put within nested function within a function as shown in the figure. Similar approach was applied for

```
--17.--BSF
   procedure BSF(signal W,f : in word8;
                 signal status_in : in word3;
                  signal bit_select : in word3; --SPECIFIES WHICH BIT TO SELECT
                 signal status    : out word3;
                 signal result    : out word8) is

     variable pos : word8;
     variable temp8 : word8;

   begin
     pos := "00000001";
     temp8 := f or (STD_LOGIC_VECTOR(SHIFT_LEFT(UNSIGNED(pos), TO_INTEGER(UNSIGNED(bit_select))) ));
     status(2) <= status_in(2);
     status(1) <= status_in(1);
     status(0) <= status_in(0);
     result <= temp8;
  end procedure BSF;
```

Figure 1.2: BSF operation

BCF with a difference that instead of OR, AND operation was used to calculate the temp8 value. Now that I had all the procedures written, I needed to "tell" to the

microcontroller that all these opcodes were not foreign entities but a part of you. For this, I declared an enumeration type "operation" in the *funtions.vhd* file. Now since these opcodes would be read from a file, I needed to make sure that whenever the "read head" of microcontroller encounters these strings the microcontroller identified them as known identifiers. For this, I wrote a function *str2op*. To make things easy, I set aside a standard string length of 5 which included all the opcodes as shown in the figure: Moreover, I also needed to "tell" the microcontroller that whenever

```
function str2op(op : in string(1 to 5)) return operation is
  begin
    if    op = "ADDWF" then return ADDWF;
    elsif op = "ANDWF" then return ANDWF;
    elsif op = "CLRF "  then return CLRF;
    elsif op = "CLRW "  then return CLRW;
    elsif op = "COMF "  then return COMF;
    elsif op = "DECF "  then return DECF;
    elsif op = "INCF "  then return INCF;
```

Figure 1.3: String to opcode function

you encounter these "new parts" of you, you initiate the corresponding procedure. For this, I wrote a Case statement which called the respective Procedures based on the opcode value as shown in the figure below. I included this case statement a concurrent statement in a new vhd file which I named *alu.vhd*.

```
architecture behavioural of ex04_alu is
  begin
    alu_function: process(W, f, op, bit_select) is
      begin
      case (op) is
        when ADDWF | ADDLW => ADDLW(W,f,status_in,status,result);
        when ANDWF | ANDLW => ANDLW(W,f,status_in, status,result);
        when CLRF          => CLRF(W,f,status_in, status,result);
        when CLRW          => CLRW(W,f,status_in, status,result);
        when COMF          => COMF(W,f,status_in, status,result);
        when DECF          => DECF(W,f,status_in, status,result);
```

Figure 1.4: Opcode Case statement

1.1.4 28.03.2024 - SATURDAY

Today's objective was to write the testbench of for the exercise. I chose my input to be of this format "ANDWF,25,0,1,0" ie Opcode, Operand1, Operand2, Status Input and Bit Select. I converted the values into std_logic_vector type in the testbench file. I ran the simulations and started debugging the errors untill I got the correct output simulation. The Result and Status Output was also written to a file using the testbench.

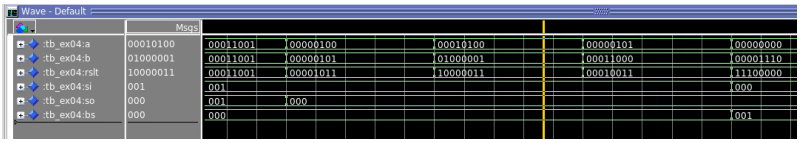

Figure 1.5: ALU output

1.2 Objective - Implementing Memory in VHDL and integrating it with ALU

1.2.1 01.04.2024 - MONDAY

I started exercise05 today. The main objective was to understand the exercise instructions and read the book chapters. The previous exercise files had to be used in this exercise as well. I copied the *functions, procedures and alu* files. Renamed the files as well as the entity names inside in order to be used for exercise 05.

1.2.2 04.04.2024 - THURSDAY

The objective was to consolidate the learning of the past few days and start writing the *Memory.vhd* file. An array had to be declared with a "depth" and "width" which would serve as memory block. Depth corresponding to the address bits and width corresponding to the data bits. I chose a Read Enable, a Write Enable and a Reset for this memory block. The address input was 7 bit long. I decided to give the address values, read enable and write enable values using a testbench file. Depending on the value of enable pins, Memory can be read, written and reset

```
entity ex05_memory is
  generic(memory_size : integer);
  port(address    : in std_logic_vector(7 downto 0);
       data_in    : in std_logic_vector(7 downto 0):= "00000000";
       memstatus_in : in std_logic_vector(2 downto 0):= "000";
       reset      : in std_logic;
       re         : in std_logic;
       we         : in std_logic;
       clk        : in std_logic;
       data_out   : out std_logic_vector(7 downto 0):= "00000000";
       memstatus_out: out std_logic_vector(2 downto 0):= "000"
     );
end entity ex05_memory;
```

Figure 1.6: Memory entity

respectively. The input file in this exercise contained three values: Opcode, W operand and Bit Select. The second operand was to come from memory this time. In this exercise, clock was also to be implemented. The main thought process went in understanding the connections between ALU and Memory.

So basically, when Memory Read is enabled, the data at the output of the memory should go into the other operand of ALU and the Status register (03h address location) bit values at that time should go into the Status Input of ALU. At the time of Write operation to Memory, the Result of ALU should go to the Data Input of Memory and the Status Output from ALU should update the Status register bit values, which is at 03h location. These Status Register bit values were the same bit values which I played with in the last exercise. There was a point to note from the exercise instructions. If the write operation happens on 03h location, then status bits are ignored. This functionality was implemented in a process, sensitive to rising edge of clock and reset. This was implemented in the *Memory.vhd* file like this:

```
architecture behavioural of ex05_memory is

       type memtype is array(0 to memory_size-1) of std_logic_vector(7 downto 0);
       signal fmem :memtype;

begin

     memory:process(all)
       begin

       if rising_edge(clk) then

         if reset = '1' then
           fmem <= (others => "00000000");
         elsif(we = '1') then
               if (address = "00000011") then

                       fmem(TO_INTEGER(UNSIGNED(address))) <= data_in;
               else
                       fmem(TO_INTEGER(UNSIGNED(address))) <= data_in;
                       fmem(3)(2 downto 0)                 <= memstatus_in;
               end if;
         elsif(re = '1') then
           data_out   <= fmem(TO_INTEGER(UNSIGNED(address)));
           memstatus_out <= fmem(3)(2 downto 0);

         end if;
```

Figure 1.7: Memory Implementation

1.2.3 13.04.2024 - MONDAY

The objective was to write the testbench for memory. The idea was that while the read enable is high, a rising edge of clock comes and reads the data using the address value at that time. The write enable is low at this instant. At the next rising edge, the address value has inscremented by one. Read enable is off and Write enable is high. The data which was read previously, now gets written to this new address. This continues untill no value remains in the input .txt file. The logic was reached after repeated simulations. The code snippet and the simulation result were recorded after debugging the errors.

```
bit_select_tb  <= STD_LOGIC_VECTOR(TO_UNSIGNED(bs_file,3));

re_tb <= '0' after 2 ns;  --STOP READING AFTER 2ns

clk_tb <= not clk_tb;     --CLK GOES FROM 0 to 1 INSTANTANEOUSLY
wait for 5 ns;            --CLK SETTLES TO 1 and RE BECOMES 0
clk_tb <= not clk_tb;     --CLK GOES TO 0

address_tb <= address_tb + "00000001";
        --MEMORY IS READ FROM THE PREVIOUS ADDRESS. NEW
        --ADDRESS POINTS TO A NEXT LOCATION.
        --WRITE OPRATION IS PERMORMED AT THIS NEW
        --LOCATION. THE READ OPERATION IN THE NEXT
        --LOOP INTERATION IS READ FROM THIS NEW
        --ADDRESS LOCATION.
we_tb <= '1' after 2 ns;
wait for 5 ns;            --WE goes to 0, clk is already0

clk_tb <= not clk_tb;     --write operation. rising edge of clk
we_tb <= '0' after 2 ns;
wait for 5 ns;            --we goes 0 (initial state)

clk_tb <= not clk_tb;     --clock goes zero
re_tb <= '1' after 2 ns;
wait for 5 ns;            --re goes 1(initial state before loop)
```

Figure 1.8: Memory testbench

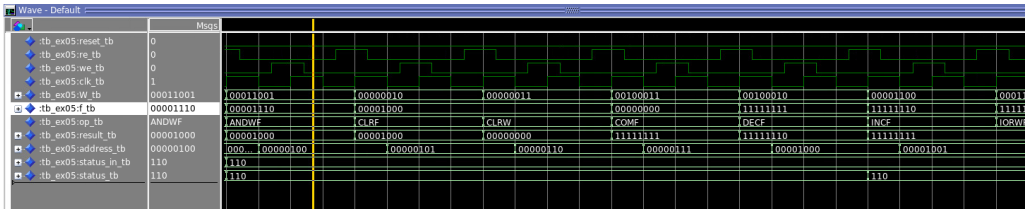

Figure 1.9: Memory Simulation

1.3 Objective - Implementing the Decoder in VHDL and integrating it with ALU and Memory

1.3.1 16.04.2024 - THURSDAY

The objective was to understand the exercise instructions and go through the suggested material. While working on the previous exercises, a good starting point for me in terms of designing the blocks was to think about the nature and source of input. In this case, the input was contained in a file and each input was a 14 bit instruction word. So the idea was that the Decoder received a 14 bit binary code, broke it into parts, made some sense out of the different parts and send this formulated "sense" to the output of the Decoder from where it was taken up by other blocks like ALU and Memory.

1.3.2 19.04.2024 - FRIDAY

The objective was to refine my understanding further, modify the existing files and start writing the decoder file. The "sense" which I talked about earlier could be achieved by designing the Decoder as a Finite State Machine with four states viz iFetch, Mread, Execute and Mwrite. This was also stated in the exercise instruction. My first step was to copy all the files of the previous exercise. Rename them and as well as the entity inside each one of them. Next, I had to think about the modifications I needed to make in the files I had so far. I had to study the datasheet again. A type of classification was to be implemented this while ie Bit type, Byte type and Literal and Control Type. For this, I included another function in my *functions.vhd* file for this exercise: I also included another enumeration type, in

```
--------------------------------------------------------------------------------
--Function which tells whether the operation is BIT, BYTE or LITERAL/CONTROL OPERATION
--------------------------------------------------------------------------------
function instructionType(instruction : std_logic_vector(13 downto 12)) return operationType is
  begin
    if instruction(13 downto 12) = "00" then
      return BYTEtype;
    elsif instruction(13 downto 12) = "01" then
      return BITtype;
    elsif instruction(13 downto 12) = "11" then
      return CTRLtype;
    else
      return NOP;
    end if;
end function instructionType;
```

Figure 1.10: Instruction Type function

the *functions.vhd* for the four different states of the State Machine which would be implemented. One observation was noted here that the funtion*str2op* was no longer needed now since now the decoder will decode the opcode from the 14 bit instruction word rather than taking one from an input file.

In the iFetch state, the data is, sort of, made available at the output ports of the decoder. For example, Opcode which goes to ALU is readied at the corresponding decoder output and likewise other data like Literal Address which goes to Memory

to fetch the data, Literal Value if a Control Instruction occurs and so on. At this point, I felt that two more components had to be included in this design. Fistly, a Work register because in this exercise, we needed to specify whether the result from ALU goes to the W register or was stored in the memory (based on the address from decoder). This functionality was controlled by the 7th bit in the byte-oriented operations. Secondly, a mux for ALU because the f-operand to ALU can now either be a direct literal in case of BCF, BSF and literal and control operations, or it can come from output of Memory based on the address received from decoder.

I also noticed that since now, in certain cases, data would be written to W register, a separate case had to be included in the *memory.vhd* file which would update the status register.

```
elsif(memstatus_wreg = '1') then  --When result is stored in w register
  fmem(3)(2 downto 0) <= memstatus_in;  --Status out from alu, which is basically connected to memstatus_in,
                                        --goes into fmem(3). This
                                        --extra condition is required
                                        --to take care of the status
                                        --out information, when the
                                        --result is stored in w register.
```

Figure 1.11: STATUS register when data is stored in W register

1.3.3 21.04.2024 - SUNDAY

Today the objective was to compile my understanding of the last two days about the Decoder in a vhdl file. Decoder after reset would go to the iFetch state. In the iFetch, an conditional statement was used for the Bit, Byte and Control & Literal type instructions. Under the respective classification, the opcode was decoded, the ALU MUX select line was chosen and the Next State of the FSM was decided. Another important point was observed here about the CLRW and CLRF Byte-type instructions. The datasheet mentioned that these operations took a direct literal, which was taken from the instruction word. As such their next state had to be Execute instead of Mread.

After the iFetch, next state was Execute, which was basically the state where ALU is operational. For this, I added an enable for ALU in my Decoder output and as well as another input in ALU entity.

The next state after Mread was Mwrite. Based on bit, byte and control type instruction, either the write enable of memory or the write enable of W register was turned on. The program counter variable was also incremented in the write operation state.

The entity for the Decoder looked like this:

```
entity ex06_decoder is
  port(instruction              : in std_logic_vector(13 downto 0);
       clk                      : in std_logic;
       reset                    : in std_logic;
       alu_literal_address      : out std_logic_vector(7 downto 0):= "00000000";  --Address of f operand in memory
       alu_bit_select           : out std_logic_vector(2 downto 0):= "000";  --bit select input to alu
       alu_opcode               : out operation; --this is an enumeration datatype described in packages.vhd and functions.vhd work files
       alu_literal              : out std_logic_vector(7 downto 0):= "00000000"; --direct literal/operand from instruction to alu for control oriented operation
       program_counter          : out integer:=0;

       alu_enable               : out std_logic;  --enables or disables alu
       we_memory                : out std_logic;  --write enables the memory
       re_memory                : out std_logic;  --read enables the memory
       sl_alu_mux               : out std_logic;  --select line of mux to alu. Used for selecting the operand to alu
       we_wregister             : out std_logic;  --write enable W operand register
       status_wreg              : out std_logic;   --used to update status out when result stored in W register
       current_state_out        : out state
       );
end entity ex06_decoder;
```

Figure 1.12: Decoder entity

ALU MUX and Wregister vhd files were also written.

```
architecture behavioural of ex06_mux is

begin

    process(all) is

    begin

        if select_line = '0' then
            f_operand <= memory_in;  --f operand of alu collects value from memory
         else
            f_operand <= literal_in;  --f operand of alu directly gets the literal, embedded in the instruction
        end if;

     end process;
---  ex06_mux.vhd   30% L16   Git-master  (VHDL)
```

Figure 1.13: ALU MUX architecture

```
architecture behavioural of ex06_wregister is

begin

     process(all) is

     begin

          if (reset = '1') then
             data_out <= (others => '0');
          elsif (rising_edge(clk) and we_wregister = '1') then
              data_out <= data_in;
          end if;

     end process;

end architecture behavioural;
---  ex06_wregister.vhd   38% L16   Git-master  (VHDL)
```

Figure 1.14: W register architecture

1.3.4 22.04.2024 - MONDAY

The objective was to write the testbench and get correct simulation result. I wrote all the components and port mapped them in the architecture of the testbench. I wrote a process which generated a clock and a while loop which kept a watch that the next instruction word is only read when execution passed through all four states. I ran into numerous errors, some syntactical while some logical.

1.3.5 24.04.2024 - WEDNESDAY

The objective was to solve the errors and get the simulation result. Figure 1.15 is a snip from my terminal showing iterative trail and error process. I was able to solve some errors but still did not get the correct result:

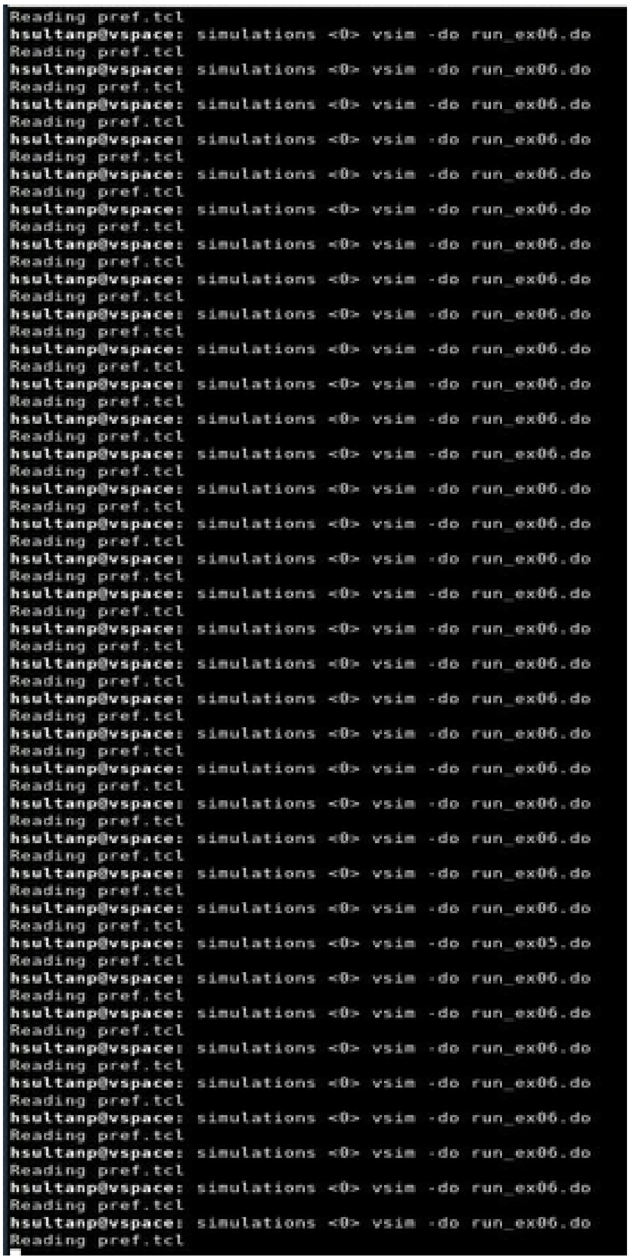

Figure 1.15: Trail and Error process

1.3.6 30.04.2024 - TUESDAY

I made an entity block diagram, shown in Figure 1.16, of the complete process so far. I colour-coded the inputs and outputs such that it is easy to track the flow.

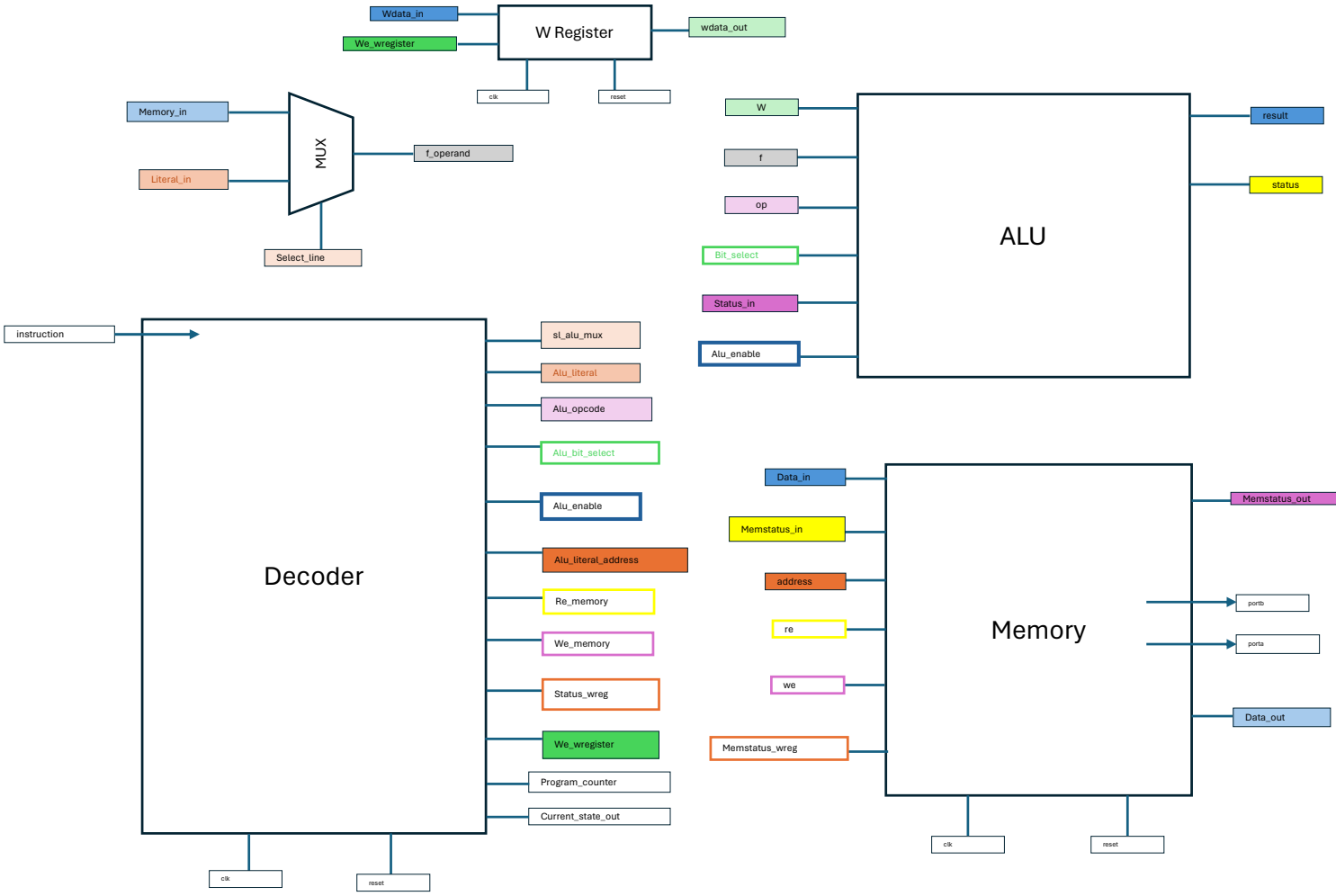

Figure 1.16: Input-Output flow of PIC components

1.3.7 11.05.2024 - SATURDAY

Today I submitted my final exercise of the course. The simulation result is shown in the figure below. I used different colours for the wave outputs to make it look more appealing and to satisfy my contentment even further (content emoji :D): Next, I

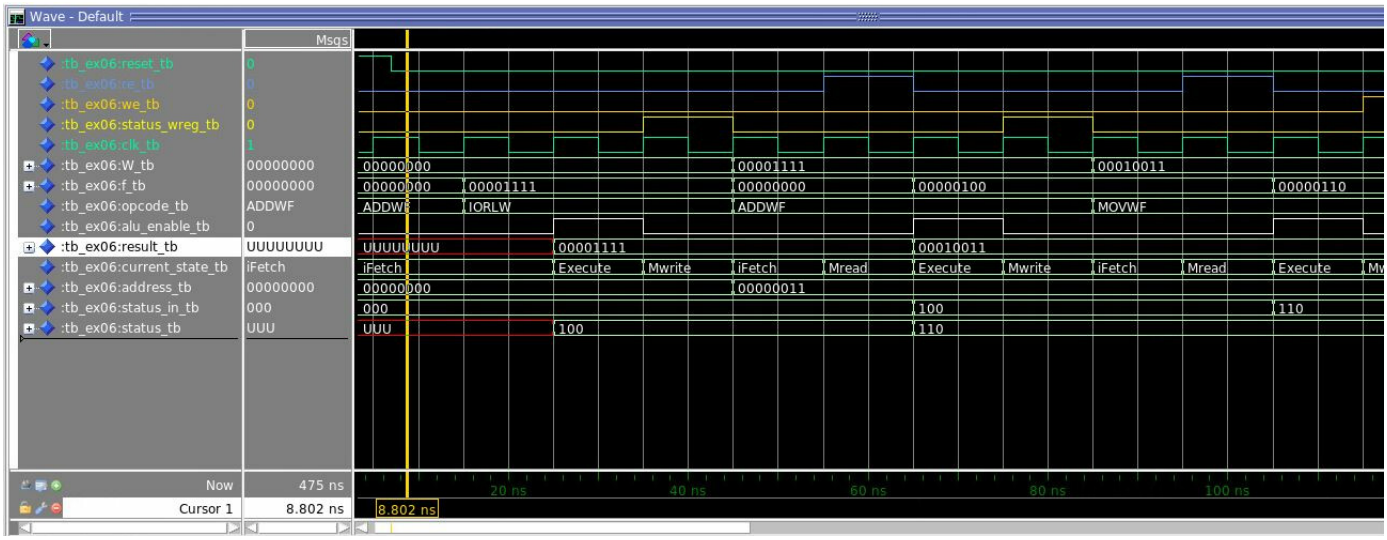

Figure 1.17: Exercise 6 simulation result

took a print of all the material needed for the project viz. Assignment Instructions PDF, Instruction.md file, PIC16F84A microcontroller project.md file. I superficially read the pdf and noticed that I have to add a few more instructions to the ALU which will be used to implement the Stack of the microcontroller. I also familiarised myself with the project folders. This is the first time I am do anything like this, I think the more I surf through these directories the more natural and easy the flow will become for me.

1.4 Objective - Implementing Stack, Program Counter and other instructions for PIC16F84A using VHDL

1.4.1 13.05.2024 - MONDAY

My final exercise was accepted with the remark "Accepted! Good job!". I kept staring at "Good Job" part for a while. :D Nonetheless. I started with *project.md* file. I read the *assignment instruction pdf* thoroughly. I think I will refer this pdf again in future. I read the PIC datasheet again and I downloaded the PicMicro Midrange reference manual. It explained the architecture and functioning of the microcontroller in a more detailed manner. This manual proved to be useful and refined my understanding further.

Next, I set up the project folder. I encountered a problem. After cloning the repository into a fresh one, the *ReportTemplate* folder was missing from the *Exerci-*

seInstruction folder. I asked for help on the Slack channel and the suggested fix was to run the command *./init_submodules.sh* in my fresh project folder. This script is written by course instructor. It worked like a refresh button in my opinion. :D The problem was solved.

Next, I encountered a problem with *makearticle* command. After running this command I was getting the prompt *Permission Denied*. I went through the Slack channel to see if any other student had faced the same problem. And voila! I ran the command *chmod +x makearticle*. This command basically gave execution rights to the file. The problem was solved. I was all done with setting up my project and Study Diary folder. I now plan to work on modifying my VHDL code to include Stack and other instructions.

1.4.2 16.05.2024 - THURSDAY

I started with understanding the functionality of Stack. I realised that I needed to modify my ALU to accommodate the CALL, GOTO, RETLW and RETURN instructions. A quick note here: I noticed that RETURN is a reserved keyword so I used RETRN for this instruction instead. I started surfing through the exercise 06 files. I realised that the *string2opcode* function in *functions.vhd* file, used to convert the string to opcode, was no longer needed after exercise 5 as I had mentioned in the diary before.

I reviewed the datasheet and noticed only the RETLW instruction seemed to require a dedicated procedure. The datasheet explained that the execution of this instruction takes the literal value (f literal) embedded within the instruction word and shifts it to the W register. I implemented this functionality in the *procedures.vhd* file. For the other three instructions (CALL, RETRN, and GOTO), a simpler approach was possible (an NOP was used). In the *alu.vhd* file, the case statement reflects this logic by utilizing the existing NOP procedure for these instructions during the execute state (ie when the ALU is operational).

Next modification was made in the *instruction2opcode* function in the *functions.vhd* file. From the datasheet, I observed instruction bits for the four operations and put the corresponding conditions in the if statement. For example, CALL is sent as an Opocde to ALU if instruction(13 downto 11) = "100" is encountered and likewise for the other additionl instructions as well.

1.4.3 17.05.2024 - FRIDAY

Writing this entry as a compilation of two full days spent in understanding the working of Stack and Program Counter, and ways to integrate it into the existing code. I searched a bit on the internet but it was not of much help and got me confused. Using ChatGPT and the datasheet provided in the repository I got clarity on the concepts but still did not feel confident. Next I turned to the PICmicro Reference Manual. I read and re-read it time and again till I developed some familiarity with the concepts. It explained the architecture of PIC microcontroller and Stack in a bit more detailed manner.

I started from the *decoder.vhd* file. In order to not make too many changes to the decoder, I figured that Stack and program counter can be written in a *process* inside the architecture of decoder. And also the Stack could be instantiated as a component in the decoder file itself. Though I would have to define the Stack entity in another separate file which will house the architecture of this Stack component. But that is for later to do.

A lot of my time was spent in understanding the operation of four additional instructions and how to implement them across the four states viz iFetch, Mread, Execute and Mwrite. CALL incremented the Program Counter before pushing its value to the Stack while GOTO did not increment. I noticed from the Reference Manual that the device Program Memory can be less than or equal to 2K: This

Note: Devices with program memory sizes 2K words and less, ignore both paging bits (PCLATH<4:3>), which are used to access program memory when more than one page is available. The use of PCLATH<4:3> as general purpose read/write bits (for these devices) is not recommended since this may affect upward compatibility with future products.

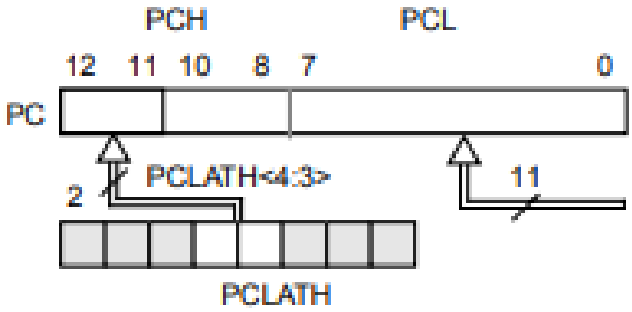

Figure 1.18: Program Counter bits information

means that the two MSB bits of Program Counter need not be taken care of while designing the Stack. And as such the Program Counter can be of 11 bits (which takes care of 2k memory addresses) and thus the width of the Stack will also be 11 bits. I also asked about it on Slack channel but did not get a response for the next two days. I decided to proceed with this simplified version, where I only have one Page in my Program Memory. It was enough to implement the Stack and Program Counter functionality.

The concept for the Program Counter and Stack implementation is described below. For better understanding, I have attach the images of code snippets as well.

For the four additional instructions, Mread state was skipped because they get the literals can be obtained directly from the instruction word. This is implemented in the *decoder.vhd fsm process block* as shown in Figure 1.19.

Generally, the Program Counter is incremented in the Mwrite state. While in the other states, Program Counter retains the value. For CALL instruction, the Execute state increments the Counter and sends it to the Top of Stack. This incremented value points to the location of the next instruction after CALL instruction. In the Mwrite state, the 11-bit address to the Subroutine (obtained from the 14-bit CALL instruction word) is fed into the Counter. From the next cycle onwards,

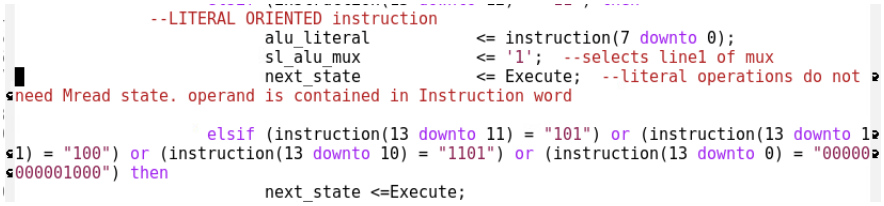

Figure 1.19: Skipping Mread for extra four operations

the execution then continues in this subroutine till a RETRN is encountered. For RETRN, Top of Stack(location of the next instruction as described above) is made available at the Stack output in Execute state. And in the Mwrite state, this value is written to the Counter. For the rest of the two instruction, the concept is simpler. For GOTO, the address location is passed to the counter in the Mwrite state. The execution continues from that address location onwards. For RETLW, the output at the stack is written to the Counter in Mwrite state.

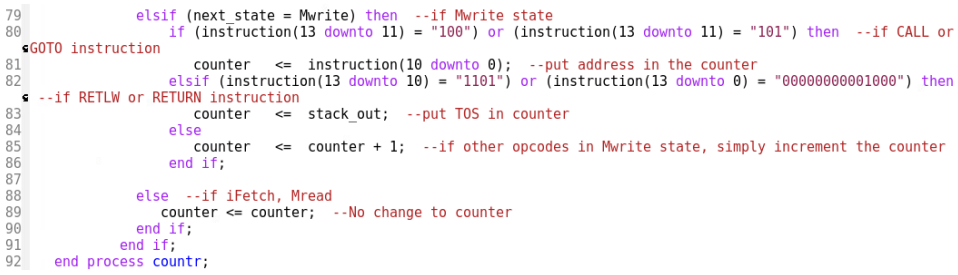

Figure 1.20: Incrementing Counter code snippet

```
elsif (next_state = Execute) then  --if Execute state
    if(instruction(13 downto 11) = "100") then  --if CALL
        push_sig  <=  '1';
        pop_sig   <=  '0';
        stack_in  <=  counter+1;  --Increment the counter and send to stack
    elsif(instruction(13 downto 0) = "00000000001000") then  --if RETURN
        push_sig      <=  '0';
        pop_sig       <=  '1';
    else  --if other opcodes
        push_sig      <=  '0';
        pop_sig       <=  '0';
    end if;
 else  --if other states apart from Execute
    push_sig  <=  '0';
    pop_sig   <=  '0';
 end if;
```

Figure 1.21: Counter and Stack operation

An 11-bit wide and 8-level deep Stack has been defined using an array in a separate *projectStack.vhd* file. The functionality has been written in a *process*. The stack pointer, index variable in this case, has a range from 0 to 7. The stack has been designed in such a way that it will take care of the pushing operation when the stack pointer is at location 7. The index wraps around after 7th location and points to location 0 again. Similarly, when the pop operation is requested at location 0 of

the stack, the index goes to location 7. This functionality can be seen from the code snippet in Figure 1.22:

```
elsif rising_edge(clk_s) then

    if push = '1' then  --if push
       stack(index) <= sdata_in;  --put data on TOS
       if (index = 7) then  --if pointer is at 7
           index := 0;      --Make it 0
       else  --if pointer is other than 7
           index := index + 1;  --increment it to make it TOS
       end if;

     elsif pop = '1' then  --if pop
       if (index = 0) then  -- if pointer is at 0
           index := 7;  --update the pointer to 7
       else  --if pointer is other than 0
           index := index - 1;  --decrement the pointer
       end if;
       sdata_out <= stack(index);  --TOS to counter
       stack(index) <= "00000000000";
     end if;
  end if;
```

Figure 1.22: Stack architecture

1.4.4 20.05.2024 - MONDAY

Next I modified the *memory.vhd* file. First, I noticed that in my existing code the address width had to be changed from, earlier 8 bits, to 7 bits. I made the necessary changes in the *memory.vhd* file and the *decoder.vhd* file. Upon studying the datasheet, I noticed that input/output ports have to be implemented which are controlled by TRISA and TRISB registers. I asked some doubts on Slack. The instructor guided me to the assignment instruction PDF which mentioned that the direction of ports are fixed and the memory addresses of PORTA and PORTB can be directly taken to output ports. Even though by now I had read the *assignment instruction pdf* many times but I think since this is all new to me, getting used to it might take some time. Nonetheless.

I next concentrated on implementing the memory with two banks, bank0 and bank1. The PIC datasheet mentions two banks for the device and also since there is a 7 bit address, the data memory ends up having two banks with 128 bytes in each bank. Moreover, the data sheet mentions that memory from 50h to 7Fh is left unimplemented as is shown in the Figure 1.23

From the same figure it can be seen that 00h address is reserved for Indirect Addressing. This means that when the address points to this location to store the data, the FSR register bits from 6 downto 0 are taken and used as address bits for the corresponding memory location. The data is then stored in this location. The bank selection in case of this Indirect Addressing is done based on the 7th bit of FSR regitser as shown in Figure 1.24.

Similar concept was used and implemented for write operation in case of Indirect Addressing. Moreover, the status bits were also observed during these read and write

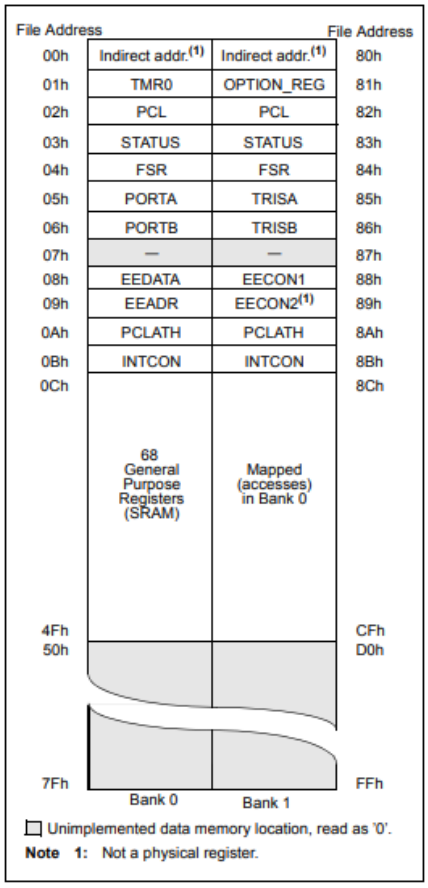

Figure 1.23: Program Memory. Banks and Registers

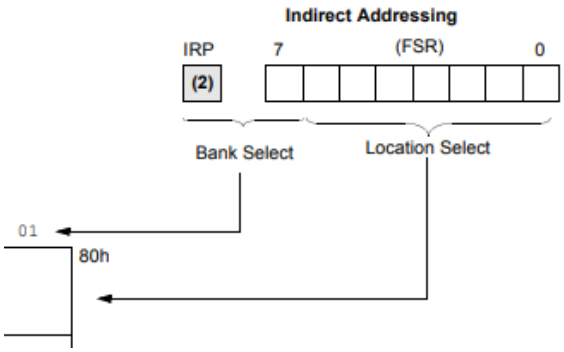

Figure 1.24: Indirect Addressing

operations. This functionality is shown by the code snippet in the Figure 1.25.

The addresses from 0Ch to 50h in the bank0 is mapped with the corresponding locations from 8Ch to CFh in bank1 ie these addresses access the same register. Also, the registers viz, PCL, FSR, PCLATH and INTCON are same in both the banks. All these points were implemented while designing the memory block. Lastly, based on the datasheet of the PIC microcontroller, the bank selection in case of Direct Addressing was done based on the 5th bit of STATUS register (03h address). This

```
--Indirect Addressing Read
-------------------------------
if(address = "0000000") then       --Read from address 00h

        if( bank0(4)(7)='0') then  --Bank 0 is Selected Based on 7thBit of FSR register
             --7 bits of FSR register(04h) is used as address to locate the data in bank0
            data_out      <= bank0(TO_INTEGER(UNSIGNED(bank0(4)(6 downto 0))));  --Data from bank 0 address is read

            memstatus_out <= bank0(3)(2 downto 0);  --STATUS register bits are read
        elsif( bank0(4)(7)='1') then  --Bank 1 is Selected
            data_out      <= bank1(TO_INTEGER(UNSIGNED(bank1(4)(6 downto 0))));  --Data from bank 1 address is read
            memstatus_out <= bank1(3)(2 downto 0);
        end if;
```

Figure 1.25: Indirect Addressing read implementation

was also implemented in *read* and *write* operation in Memory.

1.4.5 23.05.2024 - THURSDAY

From Tuesday till today, I could not spare much time for the project due to other work commitments. Today the objective was to start thinking about the testbench and get familiar with the synthesis flow from the documents provided. As far as my understanding so far is concerned I need to generate a *hex* file which will act as the instruction word input for the microcontroller, unlike in the exercises in the course, where I used to give the inputs in *csv* or *txt* files. I also studied the contents of LED blinker asm file and tinkered with *piklab* after following the instructions from the pdf file. I generated the hex file for the LED blinker asm file. I also read the *instruction.md* file for the synthesis flow. It was quite overwhelming because it seemed like a whole another game with the new scripts. A new thing called as *tcl* was introduced. It did not feel tickling at all!! I thought I would easily finish the project before time but now I was a bit worried. I had never seen such files and the instructions said that we needed to make some modifications in these files. Okayy!!

1.4.6 24.05.2024 - FRIDAY

Today, my objective was to learn to write my own assembly language program for the microcontroller and generate my own hex file. I studied the given asm code for the LED blinker again. I mostly used ChatGPT to get understand the basics of assembly language and how to write a simple file. Finally, I wrote one and named it *my_first_asml.asm*. I compiled, built and generated a hex file out of it using Piklab. I included, essentially, CALL and RETURN instructions. The provided websites on the pdf for writing the assembly code for our PIC were not helpful. They targeted some advance-level topics.

Next, I started working on writing a testbench file for my simulation. Initially, I had no idea about how to write the testbench for this project simulation. I studied the *hexfile_reader.vhd* file. I noticed the comments at the very beginning of in this file, which basically said that a function *read_ihex_file(ihex_data, memory)* was to be used to read the hex file. Upon further studying the file, I noticed that this function was declared in a package, *read_intel_hex_pack,* inside this file. I figured

that I can mention the *hexfile_reader.vhd* file in my *.do* file for this project and use the package in my testbench. In my testbench file for Exercise06, I had all the components at one place port-mapped to each other. I decided to use the same file but now with a different *Process*. I used a loop to limit and stop the simulations from running continuously. After running the testbench, I got many syntax errors in the new files and newly added code. It is 21:00 now and I call it off for the day.

2 Simulations and verification of functionality.

2.1 Objective - Debugging the errors and obtaining the simulation output

2.1.1 25.05.2024 - SATURDAY

The whole day was spent in debugging the syntax and logic errors. The most time consuming error was one with Program Memory. The error referred to the problem in *hexfile_ reader.vhd* file. It basically said that my Program Memory array is taking a value of 8199 while the range was from 0 to 1023. After much trail and error, I observed that the addresses I have used in assembly language code file were not correct for CALL and RETURN instructions. I rewrote the assembly code and generated a new hex file. Ran the simulations again. This time it worked and I got the correct simulations.

The next page haves some simulation outputs. These were obtained after further modifying the testbench, which was realised after I started the synthesis flow. A top level PIC entity was written. Since now the testbench did not have the signals which would be displayed as waves on Questa Sim, I had to learn how to display them in this scenario. This was achieved by referring to the components across files. For example *add wave -label Stack_ Output DUT:Decoder:ps:sdata_ out*. Labelling the waves in *do* file was also learnt which can also be seen in the example provided. All these edits to this diary entry was made at a later day.

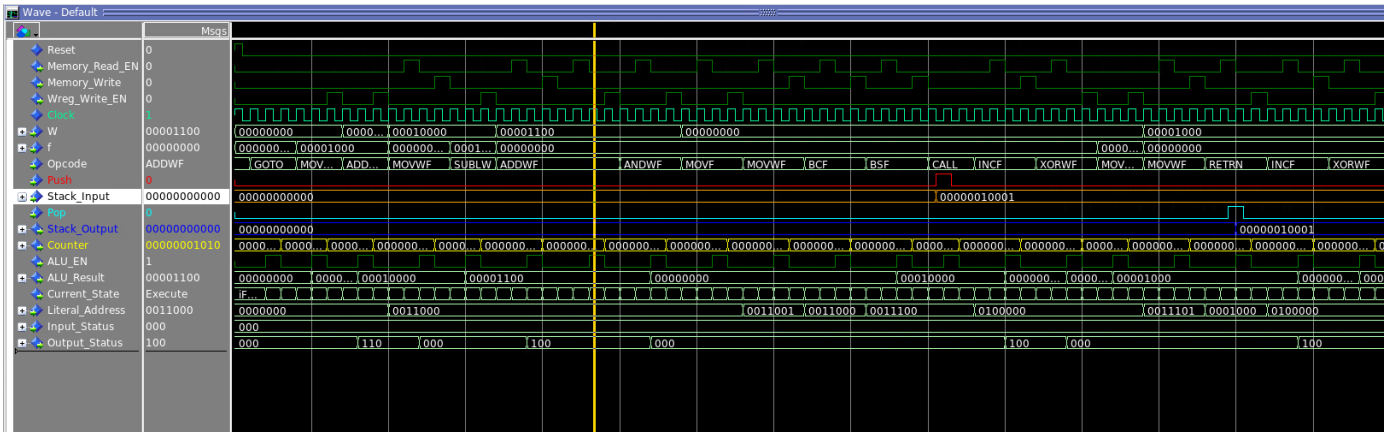

Figure 2.1: Final project simulation result

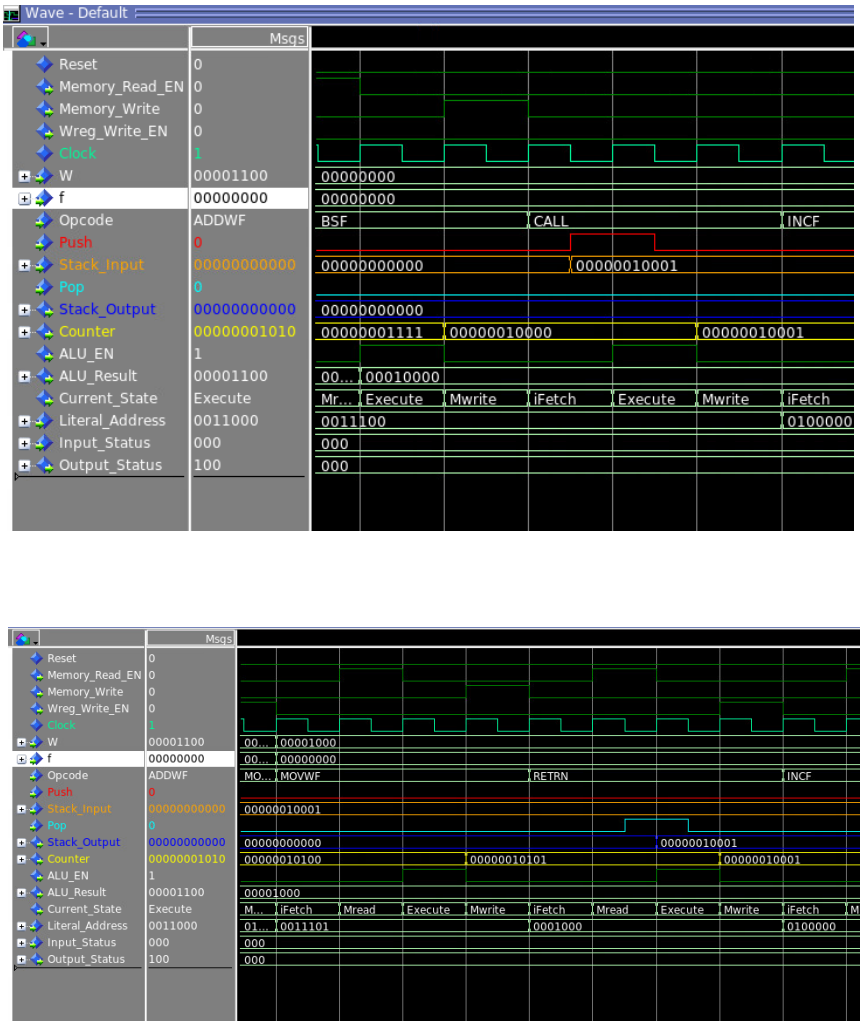

Figure 2.2: CALL and RETURN operation in action

3 Phases of Digital Synthesis

3.1 Objective - Understanding and documenting the synthesis flow

3.1.1 26.05.2024 - SUNDAY

Today, I started with synthesis part of the project. I followed the *instruction.md* file step by step. This was the second time I was reading this file. I used internet and ChatGPT to improve my understanding about many new terminologies. One thing I did was with each step I followed, I checked what changes were reflected in the *Synthesis* folder. Since I knew I had to follow the steps again when I do the synthesis with my PIC project, I figured doing this would help me understand the flow better.

After completing the synthesis flow of SPI, I noticed that all the components of PIC microcontroller and their port-mapping was done inside the testbench file whereas this testbench file was no longer needed in the syntesis process. I figured that I had to write another top-level file which would contain port-mapping between various components. I wrote *pic.vhd* file which had all the components at one place. Now that I had the PIC components in a separate file, I wrote a new testbench file for simulation purpose, which had the top-level PIC entity portmapped with this new *pic.vhd* file. I reran the simulations and got the correct output as before.

3.1.2 29.05.2024 - WEDNESDAY

Now that I had my project files ready and the simulations also came out correct, I was ready to proceed with the final synthesis process. I started off by studying all the files mentioned in the *instruction.md* file. Like in the case of SPI which had verilog files contained in separate folder inside the *synthesis* folder, I also created a separte folder for the project VHDL files and named it *picvhdlfiles*. Now I had to somehow tell the synthesiser about the location of the VHDL files. I found the environment variable *ROOTPATH* in the *configure* file and included the path to all the project VHDL files there.

After running this configuration script, *common_vars.tcl* files gets generated at this path *./synthesis/flow/scripts/common_vars.tcl*. This file sets the environment variables for the synthesis. Upon studying this file, I observed that the last two variables viz SDCFILE and CPFFILE did not have syntax colouring. I right away

```
#Paths to verilog files
LINKEDVERILOG=""

#Paths to VHDL files
LINKEDVHDL="
    ${ROOTPATH}/picvhdlfiles/project_alu_functions.vhd \
    ${ROOTPATH}/picvhdlfiles/project_alu_procedures.vhd \
    ${ROOTPATH}/picvhdlfiles/project_decoder.vhd \
    ${ROOTPATH}/picvhdlfiles/project_stack.vhd \
    ${ROOTPATH}/picvhdlfiles/project_alu.vhd \
    ${ROOTPATH}/picvhdlfiles/project_memory.vhd \
    ${ROOTPATH}/picvhdlfiles/project_mux.vhd \
    ${ROOTPATH}/picvhdlfiles/project_wregister.vhd \
    ${ROOTPATH}/picvhdlfiles/pic.vhd"
```

Figure 3.1: Link to VHDL files made in Configure script

knew that something is off with the way I have linked my VHDL files in the *configure* file. And I was right. There was a slight syntactical error. I rectified it and everything seemed in order. I noticed that all my files were now listed here but I still had the top module name as "spi_slave". I made the corresponding changes to the configure file and reran the flow so far.

```
set TOPMODULE "pic"
set VERILOGFILES ""
set VHDLFILES " /home/hsultanp/ELEC-E9540/PIC16F84A/synthesis/source/project_alu_functions.vhd /hom
e/hsultanp/ELEC-E9540/PIC16F84A/synthesis/source/project_alu_procedures.vhd /home/hsultanp/ELEC-E95
40/PIC16F84A/synthesis/source/\ /home/hsultanp/ELEC-E9540/PIC16F84A/synthesis/source/project_decode
r.vhd /home/hsultanp/ELEC-E9540/PIC16F84A/synthesis/source/\ /home/hsultanp/ELEC-E9540/PIC16F84A/sy
nthesis/source/project_stack.vhd /home/hsultanp/ELEC-E9540/PIC16F84A/synthesis/source/project_alu.v
hd /home/hsultanp/ELEC-E9540/PIC16F84A/synthesis/source/project_memory.vhd /home/hsultanp/ELEC-E954
0/PIC16F84A/synthesis/source/project_mux.vhd /home/hsultanp/ELEC-E9540/PIC16F84A/synthesis/source/\
 /home/hsultanp/ELEC-E9540/PIC16F84A/synthesis/source/project_wregister.vhd /home/hsultanp/ELEC-E95
40/PIC16F84A/synthesis/source/pic.vhd"
set SDCFILE "/home/hsultanp/ELEC-E9540/PIC16F84A/synthesis/constraints/timing_constraints.sdc"
set CPFFILE "/home/hsultanp/ELEC-E9540/PIC16F84A/synthesis/constraints/power_domains.cpf"

---  common_vars.tcl   Bot L14    (Tcl)
```

Figure 3.2: Common_vars file

After sourcing the variables through the *common_vars.tcl* file in genus terminal, the next step was to set up the technology parameters. This step also imported timing constraints from SDC files and since the instructions pointed out that we needed to change *timing_constraints.sdc* file, I decided to study its contents before running *tecnology setup* command in the genus terminal.

Basically, through this file the synthesiser is given clock information, unlike in the course exercises wherein the clock information was given through a *process* inside the testbench file. I noticed that the *timing_constraint* file had two clocks defined, whereas the project PIC design only needed one clock, therefore I removed the other clock named *SCLK_P 10.0*. I next removed the spi sclk. Under clock group, only DSP master was kept. Also clock latency for sclk was removed. For I/O constraints, since my input to PIC is only an instruction word, I kept an input delay at that port, rest all were removed. Similarly, pertaining to the master clock I set delays for my three outputs viz *porta_pic*, *portb_pic* and *pc_pic*.

Next, I ran the command to read *lef* files. These files basically contain the data pertaining to the physical design of the chip and these are read through the functions

used in these command.

Then I ran the commmand for the *design files*. Since, I had no verilog files, I ran the if clause for VHDL files. I encountered very strange errors. I believe I finally witnessed the difference between a VHDL code which can be synthesised and a VHDL code which can be simulated.

The strangest error was that the synthesiser was not able to find the linked VHDL files. After much brain hammering, the problem was that I had not placed the files in a proper order while linking the files. For example, the top-level *pic.vhd* file had to be be put at the bottom. The final order is shown in Figure 3.1. I got a few more repetitive errors. Some of the different types are shown in the figures 3.3, 3.4. I called it off for the day.

```
Error   : Missing token. [VHDLPT-672] [read_hdl]
        : subprogram specification requires ';', read keyword SIGNAL in file '/h
ome/hsultanp/ELEC-E9540/PIC16F84A/synthesis/source/project_alu_procedures.vhd' o
n line 148, column 19.
                  signal status_in : in  std_logic_vector(2 downto 0);

                  |
Error   : Declaration not allowed here. [VHDLPT-775] [read_hdl]
        : Signal declaration not allowed in package body declarative part in fil
e '/home/hsultanp/ELEC-E9540/PIC16F84A/synthesis/source/project_alu_procedures.v
hd' on line 148, column 19.
        : Invalid or unsupported VHDL syntax is encountered.
                  signal status_in : in  std_logic_vector(2 downto 0);

                                     |
Error   : Unexpected construct. [VHDLPT-630] [read_hdl]
        : Expecting a declarative item in file '/home/hsultanp/ELEC-E9540/PIC16F
84A/synthesis/source/project_alu_procedures.vhd' on line 148, column 38.
        : Invalid or unsupported VHDL syntax is encountered.
                  signal status    : out std_logic_vector(2 downto 0);
                  |
```

Figure 3.3: Errors while synthesising

```
   procedure COMF(signal W,f      : in std_logic_vector(7 downto 0);
   |
Error   : Missing subprogram body. [VHDLPT-666] [read_hdl]
        : For procedure 'default.project_alu_procedures.COMF[std_logic_vector, s
td_logic_vector, std_logic_vector, std_logic_vector, std_logic_vector]' in file
'/home/hsultanp/ELEC-E9540/PIC16F84A/synthesis/source/project_alu_procedures.vhd
' on line 48, column 4.
        : Invalid or unsupported VHDL syntax is encountered.
```

Figure 3.4: Errors while synthesising

3.1.3 07.06.2024 - FRIDAY

Since past couple of days, I was modifying parts of VHDL to make it synthesisable. For example, I learnt that synthesiser did not accept the subtypes. I had to replace them all. Also I had defined multiple signals in one statement, for example in ALU, W and f port were declared in entity in a single statement. Synthesiser did not accept this. But I still have some unresolved errors. I could not keep a track of the version which I have followed while writing VHDL. Almost all the errors are because of the mismatch between the tool and the VHDL version used.

I aimed for a grade 5 in this course as I gave it my sweat and blood right from the inception of this course. Unfortunately, I was not able to finish it completely,

```
subtype word9 is std_logic_vector(8 downto 0);
subtype word8 is std_logic_vector(7 downto 0);
subtype word7 is std_logic_vector(6 downto 0);
subtype word5 is std_logic_vector(4 downto 0);
subtype word4 is std_logic_vector(3 downto 0);
subtype word3 is std_logic_vector(2 downto 0);
subtype word1 is std_logic;
```

Figure 3.5: Removed subtypes for synthesis process

even though, I feel I have done a lot in this course already and learnt something substantial.

4 Chapter 2