Study Diary on implementation of internal architecture of PIC16F84A

Faizan Ahmad Khan

31.05.2024

Aalto University
School of Electrical
Engineering

Contents

1 Introduction

I am writing this study diary to share my thought process when designing the microcontroller unit of PIC16F84A. My journey of learning started with the 6 basic VHDL coding assignments. The first 3 assignments were important in developing my basics in VHDL programming since I had no prior experience in VHDL programming. While the last three assignments provided the crucial knowledge of developingthe PIC16F84A microcontroller unit. I have divided the chapters into the different stages required in implementing the architecture of PIC MCU. The implementation of PIC MCU starts with the design and development of synthesizable VHDL code for PIC16F84A which is explained in chapter 2. After developing the VHDL code, it is also important to test the code with dataset to verify whether the code is working as intended. The verification of code can be achieved by the development of testbench which would be explained in chapter 3. Lastly, the RTL design is extracted from a functional RTL code and then the RTL design is synthesized. The process of synthesis of RTL schematic of PIC16F84A will be explained in the chapter 4. chapter 5, concludes my overall learning experience throughout the project development.

2 Design and development of VHDL code

In this chapter, I will try to explain my thought process while designing the vhdl code. I took the help of internet(chat gpt) and [1] to remove syntax errors and improve my code. I would also like to mention the name of one of the teaching assistants Aleksi Korsman who helped me the most in vhdl programming. For example, in the very first exercise session where we had to develop a multiplexer and I had used wait statements in my process statements. Aleksi told me that although my code was working but it wouldn't be synthesized and instead of wait statements. I should include my signals in the sensitivity list of the process statement. And wait statements should can only be used in the testbench. I would to try explain each segment of my code as much as I can based on the architecture of the PIC shown in Figure 2.1.

The three important segments of PIC microcontroller are the ALU where it performs all the operations then is the data memory which includes SRAM (General Purpose Registers) and special function registers. And then the last and important segment is the decoder which is synchronous state machine divided into four stages iFETCH, MREAD, EXECUTE and MWRITE. I will explain the development of vhdl code for each segment in more detail. Beside this the PIC microcontroller includes a number of register, multiplexers, a counter and a stack. Currently, I wouldn't be discussing stack since I do not have enough time to include it in my code. At the end of the chapter I will explain the wiring of these components that was implemented in my top module.

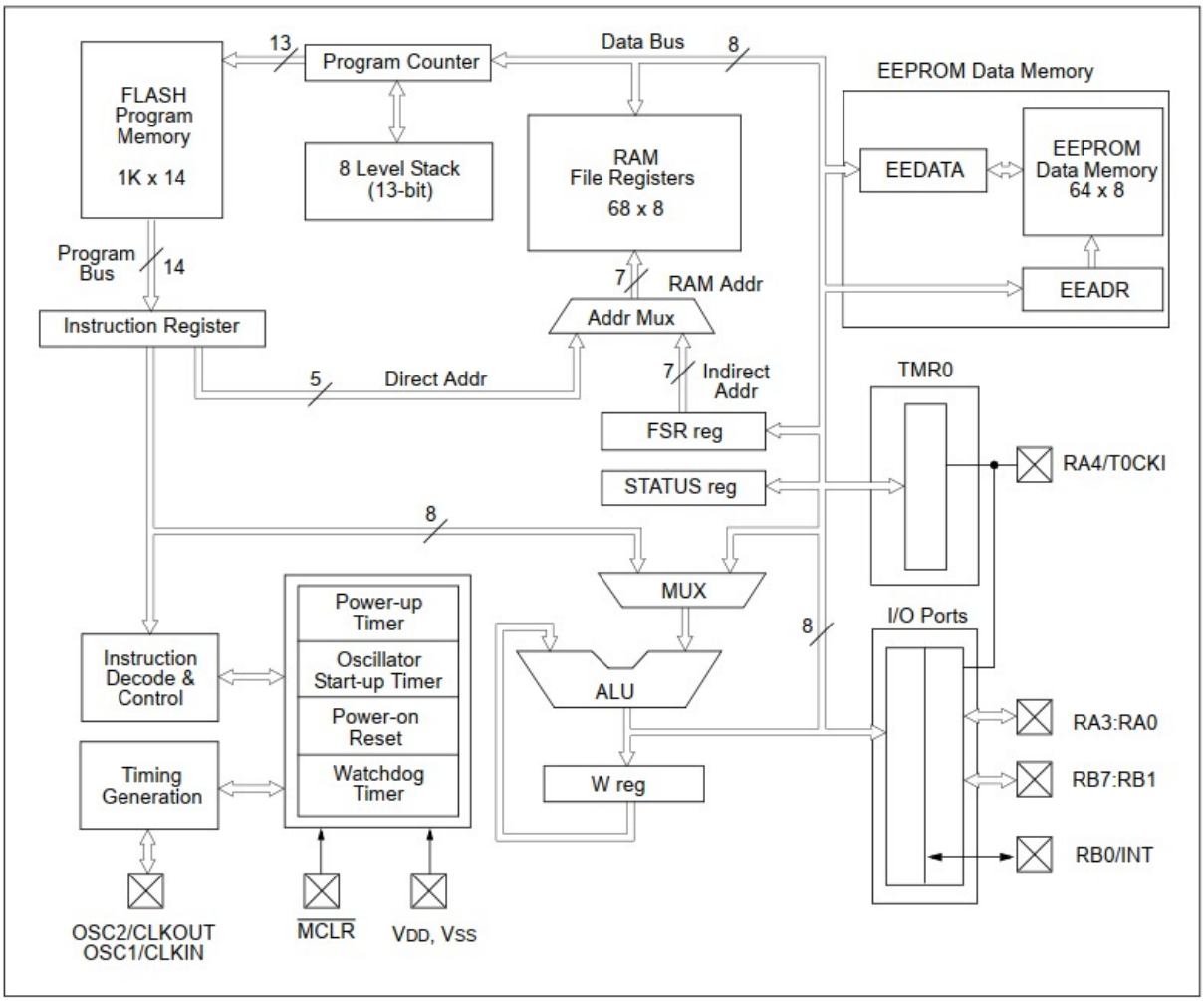

Figure 2.1: PIC16F84A Architecture

2.1 Implementation of ALU

The first step towards the project was implementation of the ALU and this was the first time when I had to read the PIC datasheet and reference manual. First I tried to understand the architecture of ALU which is best explained by the Figure 2.2.

The module of ALU included two input ports named W and F which in the image above comes for accumulator (W register) and F comes from either the value stored in memory or directly/literally from the instruction provided to the decoder. Similarly, an output port by the name of result will either store result of ALU in memory or accumulator as shown in the picture above. The selection of F input and output storage based on d bit of instruction will be discussed in the upcoming section 2.3. Moreover input and output port of status bit is also included in the ALU module. An input port of enumaration type for operations was implemented in package which included byte, bit and almost all of the literal operations. For implementation of the different instructions in ALU the instruction description provided

in PIC datasheet shown in Figure 2.3.

This was implemented in VHDL using case operator which based on the operation enum type would decide which specific instruction to execute. Status bit for the instructions that would raise zero, digit carry and carry flag output status bits were changed on whether the result of the instruction was zero, had carry been generated on the 4th bit or last bit of 8 bit result. The implementation of these flags was done using procedures in vhdl which were called in the process statement where the operation is performed. The procedure will generate the flag i.e. return 1 if result is zero or carry is generated on 4th or 8th bit.

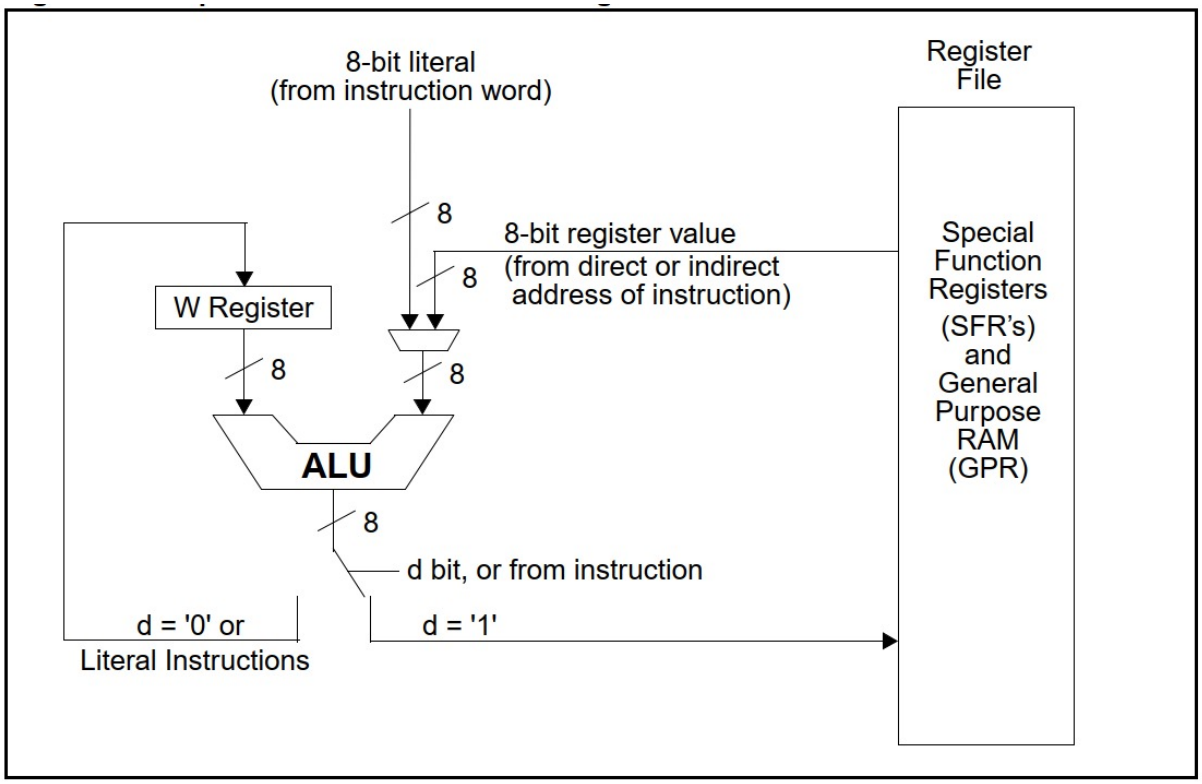

Figure 2.2: ALU Architecture

7.1 Instruction Descriptions

ADDLW	Add Literal and W
Syntax:	[*label*] ADDLW k
Operands:	0 ≤ k ≤ 255
Operation:	(W) + k → (W)
Status Affected:	C, DC, Z
Description:	The contents of the W register are added to the eight-bit literal 'k' and the result is placed in the W register.

ADDWF	Add W and f
Syntax:	[*label*] ADDWF f,d
Operands:	0 ≤ f ≤ 127 d ∈ [0,1]
Operation:	(W) + (f) → (destination)
Status Affected:	C, DC, Z
Description:	Add the contents of the W register with register 'f'. If 'd' is 0, the result is stored in the W register. If 'd' is 1, the result is stored back in register 'f'.

ANDLW	AND Literal with W
Syntax:	[*label*] ANDLW k
Operands:	0 ≤ k ≤ 255
Operation:	(W) .AND. (k) → (W)
Status Affected:	Z
Description:	The contents of W register are AND'ed with the eight-bit literal 'k'. The result is placed in the W register.

ANDWF	AND W with f
Syntax:	[*label*] ANDWF f,d
Operands:	0 ≤ f ≤ 127 d ∈ [0,1]
Operation:	(W) .AND. (f) → (destination)
Status Affected:	Z
Description:	AND the W register with register 'f'. If 'd' is 0, the result is stored in the W register. If 'd' is 1, the result is stored back in register 'f'.

BCF	Bit Clear f
Syntax:	[*label*] BCF f,b
Operands:	0 ≤ f ≤ 127 0 ≤ b ≤ 7
Operation:	0 → (f<b>)
Status Affected:	None
Description:	Bit 'b' in register 'f' is cleared.

BSF	Bit Set f
Syntax:	[*label*] BSF f,b
Operands:	0 ≤ f ≤ 127 0 ≤ b ≤ 7
Operation:	1 → (f<b>)
Status Affected:	None
Description:	Bit 'b' in register 'f' is set.

BTFSS	Bit Test f, Skip if Set
Syntax:	[*label*] BTFSS f,b
Operands:	0 ≤ f ≤ 127 0 ≤ b < 7
Operation:	skip if (f<b>) = 1
Status Affected:	None
Description:	If bit 'b' in register 'f' is '0', the next instruction is executed. If bit 'b' is '1', then the next instruction is discarded and a NOP is executed instead, making this a 2TCY instruction.

Figure 2.3: Instruction Description

2.2 Implementation of Data Memory

I try to implement the data memory block of PIC microcontroller which consists of two banks 0 and 1 controlled by a register bank select bit (RP0). The data memory block is divided into special function and general purpose registers. Intially, I am trying to implement the registers in bank 0 i.e. the status register (special function register) and 68 bytes of SRAM.

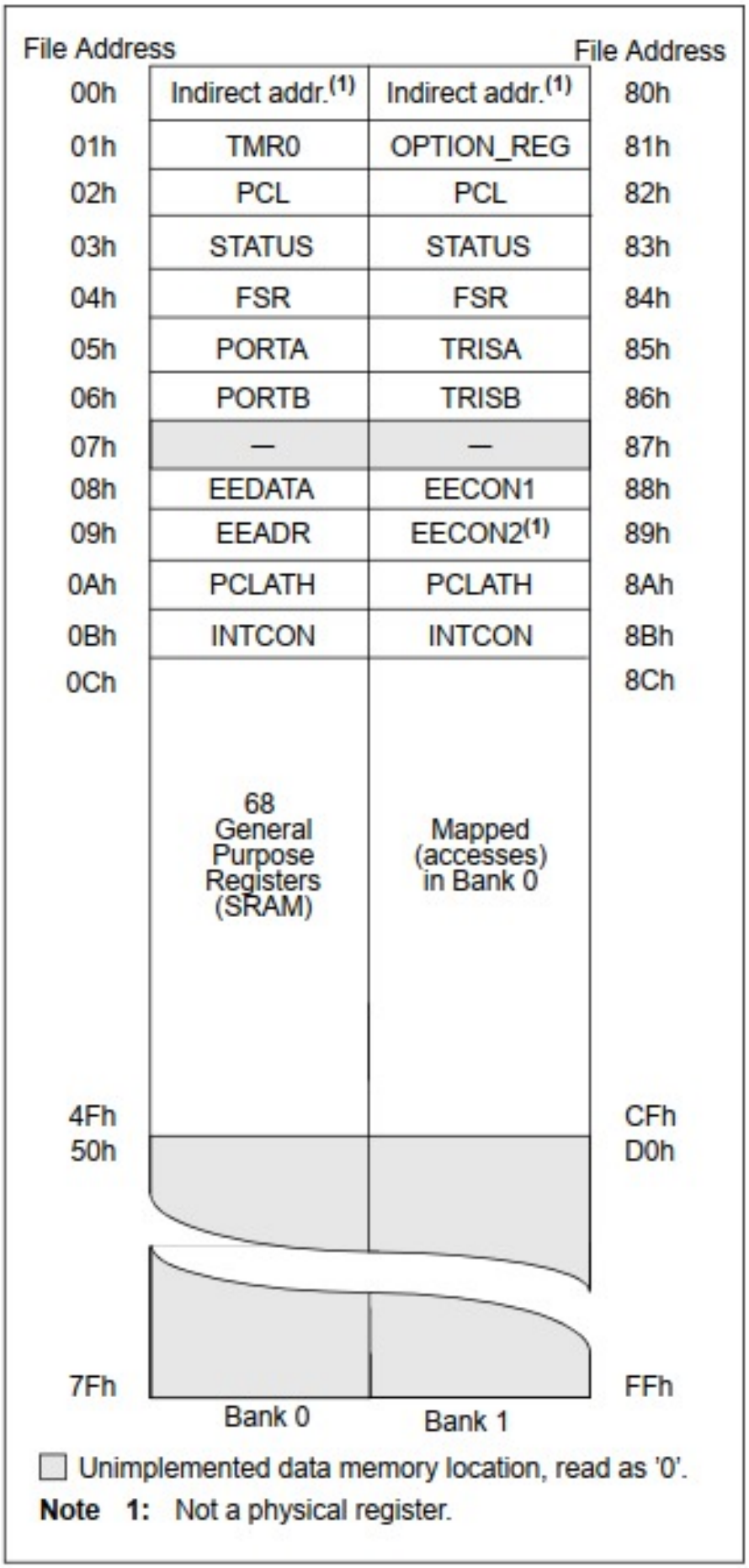

Figure 2.4: Structure of Data memory block in PIC microcontroller

2.3 Implementation of Decoder

Iniatially, I try to implement a four stage decoder consisting of iFETCH, MREAD, EXECUTE and MWRITE. I implemented this using two process statements in vhdl since process statements works in parallel to eachother. The state machine process ensures the transistion from current stage to next stage is synchronous i.e changes on positive edge of the clock. And the instruction decoder process is where the entire working logic of decoder is implemented for the four stages. I implemented the logic of decoder such that when the microcontroller is powered ON (i.e. reset). The decoder will start with the fetch stage. The fetch is implemented such that it will decode the instruction by extracting the 5 MSB bits from the instruction which contains the opcode. Based on the extracted opcode it will decode which operation is to be performed. Based on the type of the opcode i.e whether it is byte-oriented, bit-oriented or literal operation, it will extract the 7-bit address bits carried by byte-orientedand and bit-oriented operations and would extract the 8-bit imediate value for ALU operations from literal and control operations. The logic for fetch stage was derived from the information provided in PIC datasheet regarding the instruction format and can be seen in Figure 2.5.

Based on the opcode, if the instruction involves reading from memory i.e. contains address bits will jump to MREAD stage otherwise the state will directly change to EXECUTE. In MREAD stage, the decoder will read value from the RAM in data memory block based on the specific address bits extracted from the instruction bits and the control signals for the RAM would be such that the read enable signal (ren) will be high and the write enable signal (wen) will be low.

In the execution stage EXECUTE, the decoder will decide that based on the opcode that which operation should be perfromed by the ALU. During this stage, it also sends control signal to the data selection MUX which provides f operand to the ALU from one of the two sources i.e. direct/literal value from the instruction or a value stored in RAM. This control signal for the data selection MUX is provided on the basis of the operation to be performed by the ALU i.e '0' in case of bit/byte operations and '1' in case of literal operations.

Lastly, the WRITE stage decides where to write the ALU result i.e. in RAM or the accumulator (W register) based on the d signal and the specific operation. The ALU result for bit operations is always stored in the RAM. Similarly, the ALU result for literal operations is always stored in W register. While for byte operations, the ALU result storage depends on the d bit extracted from its instruction during iFETCH stage. This four stage state machine works fine if the instructions are directly fetched from the program memory to the decoder.

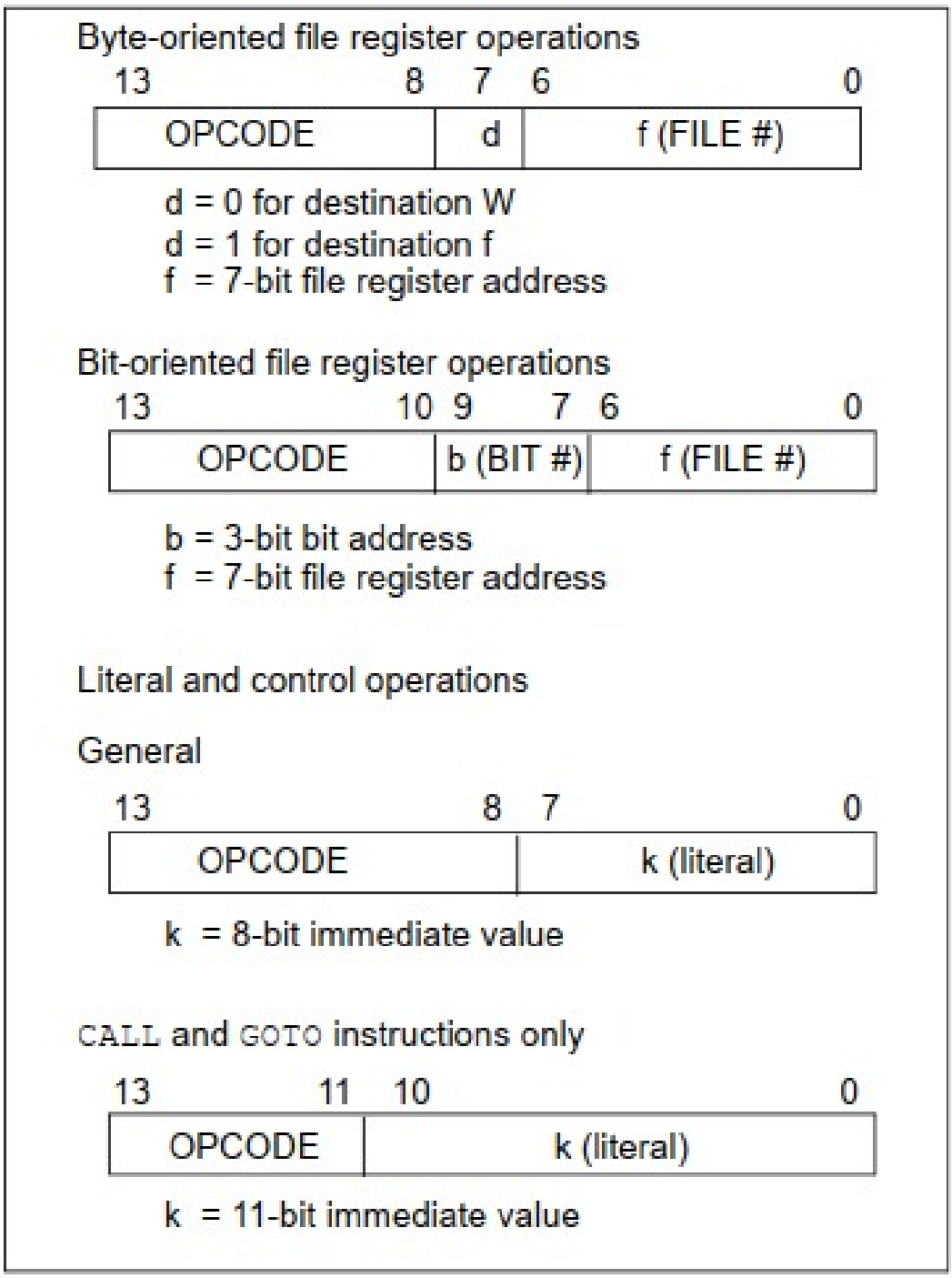

Figure 2.5: General format for Instructions

Although, the PIC instruction cycle should be 200ns as mentioned on the first page of PIC datasheet i.e. it should consists of four stages and each stage takes 1 clock cycle time to complete. One clock cycle time for PIC is 50ns if the external oscillator is oscillating at 20 MHz. But this is not the case, as can be seen in Figure 2.1, the instructions from program memory are first fetched into instruction register and then to the instruction decoder. The fetching of instructions into the

instruction registers creates a delay of one clock cycle when it reaches the decoder as the register will transfer data from input to output on positive edge of the clock. Due to this delay the instructions are not received by the decoder in fetch stage as can be seen in Figure 2.6.

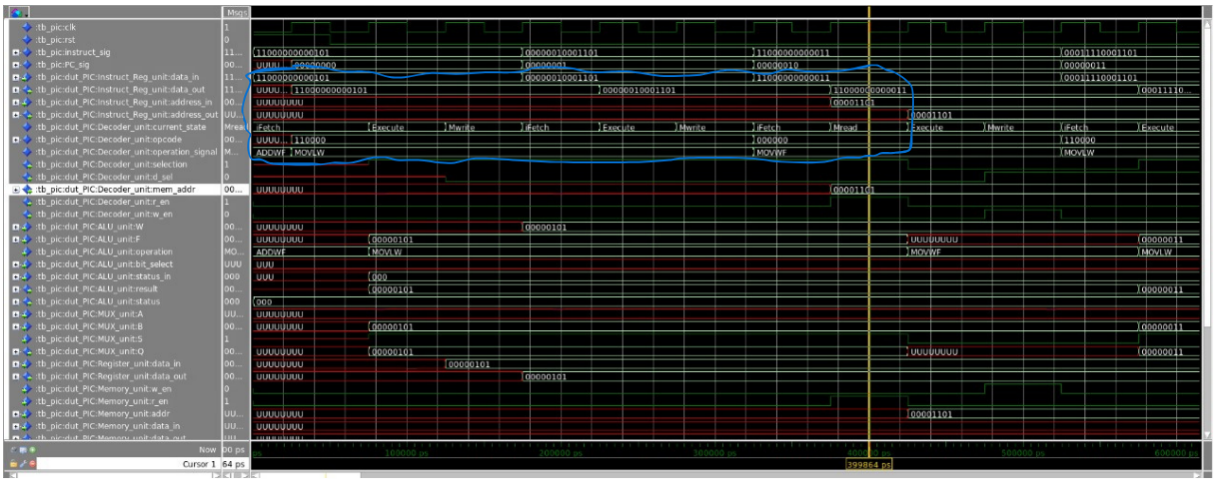

Figure 2.6: Waveform showing delay in instruction caused by instruction register

To cope with the delay produced by the instruction register, a new stage before the iFETCH statement by the name of DELAY was introduced. Now, the stages have increased to 5 stages but the instructions are now available at the fetch stage. But it is also observed in Figure 2.1 that the addresses extracted by the decoder used for reading and writing values from RAM also passes through the instruction register. A similar delay is caused in addresses and the addresses extracted in iFETCH stage are not available in the READ stage due to delay caused by instruction registers which can be seen in Figure 2.7.

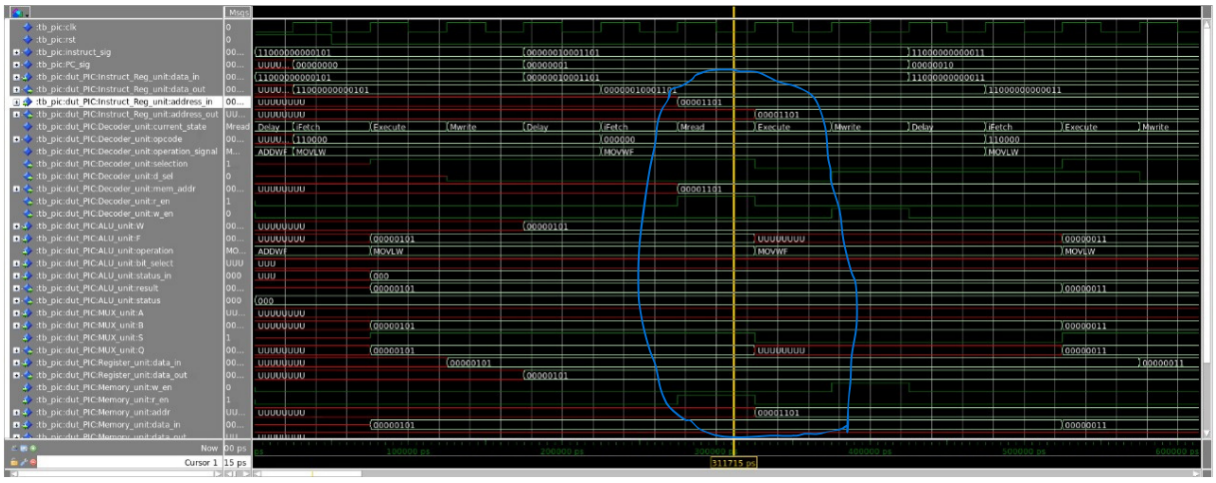

Figure 2.7: Waveform showing delay in address caused by instruction register

Therefore, the operations that require MREAD stage would have an additional DELAY_MREAD stage in order to ensure that address extracted in iFETCH stage

is available in MREAD stage. Now a single instruction may take atmost 6 clock cycles instead of the requirement of 4 clock cycles but I couldn't think of any other method to cover up the delay produced by the instruction register. The only other method was to implement an asynchronous instruction register that won't wait for the edge of the clock to change the value but again this register should be synchronous for PIC microcontroller.

2.4 Implementation of Counter

The next important component is the counter which controls the instructions coming from the program memory. The counter is designed such that it will increment after the occurance of the last stage i.e MWRITE. The 13 bit counter points to the address of the program memory and after every increment it will point to the address of the next instruction. Intially, I was also facing delay in the value received on the counter port which was used in the testbench to control the instruction flow. But after thinking for somewhile, I realized that I am incrementing the counter and storing in signal which is then converted to std logic type in the next step and supplied to the PC port. But the value being transfer to PC port after two steps was creating a delay. Therefore, instead of storing the incremented value in the pc_reg signal and then transfering it to the PC port. I made changes in the code by first directly transfering the incremented value of counter to PC port and then in second step the incremented value is stored in pc_reg. Both of the changes are shown in Figure 2.8.

```
-- counter increments after each instruction reaches Mwrite stage
if state = Mwrite then
-- pc_reg <= unsigned(pc_reg) + 1;
-- pc <= std_logic_vector(pc_reg);
  pc <= std_logic_vector(unsigned(pc_reg) + 1);
  pc_reg <= pc_reg +1;
end if;
```

Figure 2.8: Changes in code to remove delay in counter

3 Testing the VHDL Code with a Testbench

In this chapter, I will try to explain the implementation of testbench for testing my vhdl code and verifying my logic. The testbench that I build basically reads a number of instructionss from a text file. The code for reading a text file is implemented in the read_file process which was relatively easy since I have been building it from assignment 2. Beside this, I created clock_gen process which generates the clock and reset_gen that generates reset. Initially, I implemented the reset inside the read_file process but then to add clearity in my code I implemented it in a seperate process statement since process are concurrent statements execute in parallel to eachother. The important thing the entire testbench was the control of instructions read by the testbench from the text file which I would like to refer as the program memory of PIC microcontroller. Now if the text file is the program memory of PIC microcontroller and if we check out the Figure 2.1 in **??** which shows the architecture of PIC, then we can see that the instructions fetched from the program memory to the instruction register are controlled by the program counter. This was achieved with the following piece of code shown in Figure 3.1.

```
instruct_sig <= to_stdlogicvector(instruction_val);

-- Wait for the PC to change before reading the next instruction
wait until PC_sig /= last_pc;
last_pc <= PC_sig;
```

Figure 3.1: Testbench code for controlling instructions

In the above code, the last_pc is intiallized to 0 and pc_sig is the signal coming from the program counter. Now the code will wait until the pc_sig changes its value i.e. increment to 1 and then in the next step the last_pc will now retain the value of 1 and the code will read the next instruction. The code will again wait until the value of pc_sig changes to 2 and so on. The instructions that I used to test the code, starts with MOVLW operation since the W register is empty so the first thing is to preload a value in the accumulator. And the last instuction is of CLRW which will clear the accumulator of any previous stored value before a new instruction set is loaded into the program memory i.e. text file with completely new instructions.

4 RTL schematic generation and Synthesis

In this chapter, I will briefly described the steps that I had gone through for implementing synthesis using Cadence Genus digital synthesizer. The first thing was reading the instructions given under the synthesis directory and try to follow it step by step. The first thing the instructions mentioned were to edit the timing constraints (.sdc file). In the sdc file which contained the timing constraints, I first changed the clock period to 50ns as it is mentioned on the first page of data sheet that the clock input is 20 MHz (50ns). Moreover, I kept the latency to 1 ns and changed the input transition time to 5 ns (10% of 50 ns) as it was previously set to 1 ns for 10 ns clock period. The capacitive load for all output pins was set to 50 pF as mentioned on page 56 of the datasheet under the title of Timing conditions. In this way, the entire file was edited according to our code for PIC microcontroller.

After making the nescassary changes in timing constraints, I added the path to my vhdl files in the configure file which were added into the common_var tcl script generated after running the configure file. The common_var.tcl script file contained the variables used in the main synthesis.tcl script. Next by running ğenus -common_uiċommand as mentioned in the instructions, started the genus command line. Then I run each command mentioned in the synthesis.tcl script file step by step. Intially, I couldn't read my vhdl files using the script mentioned for reading vhdl files although the verilog script was working perfectly fine and was able to read the verilog files. I got multiple errors for it. Then I searched for each of the error by typing it on google and I got to know that the error the synthesizer was giving i.e. Ërror : Missing token. [VHDLPT-672] [read_hdl] m̈eant that the synthesizer could not read the IEEE library. So, I included this command: s̈et_db hdl_vhdl_read_version 2008 ∴ But I was still getting errors and then realized that I had first compile and read the packages I am using in vhdl files and then read the files itself. I was able to achieve this by modifying the script with the command lines seen in Figure 4.1.

```
# Defining the path file for the package files used in my other vhdl files
set type_package_path "/home/fkhan/ELEC-E9540/PIC16F84A/vhdl/type_package.vhd"
set state_package_path "/home/fkhan/ELEC-E9540/PIC16F84A/vhdl/state_package.vhd"

# Separate package files from other VHDL files
set PACKAGEFILES [list $type_package_path $state_package_path]

# Check if there are VHDL files to read
if {[llength $PACKAGEFILES]} {
    puts "Reading in package files"
    foreach file $PACKAGEFILES {
        puts "Compiling package: $file"
        read_hdl -language vhdl $file
    }
} else {
    puts "No package files to read"
}

# Check if there are other VHDL files to read

if {[llength $VHDLFILES]} {
 puts "Reading in vhdl files"
  foreach file $VHDLFILES {
        puts "Compiling file: $file"
        read_hdl -language vhdl $file
        set_db hdl_vhdl_read_version 2008
    }

} else {
  puts "No vhdlfiles to read"
}
```

Figure 4.1: Modification in vhdl read script

5 Conclusion

There is alot that I could still would have done in the project. But still I will regard it as one of best learning experience. I have learned alot in this course. I have mastered git, emacs text editor along with VHDL programming.

Bibliography

[1] P. J. Ashenden, *The designer's guide to VHDL*. Morgan kaufmann, 2010.