Comparison of Different Micro-architectures for a Subset of 8085

VLSI Design Lab Project

Submitted in partial fulfilment of requirements of the degree of

Master of Technology
Electronic Systems

By

Gourav Tilwankar - 203070051
Harsh Paryani - 203070052
Tanuj Kumar - 203070059
Deepak Chand - 203070060

under the guidance of
Prof. Sachin Patkar

Department of Electrical Engineering
INDIAN INSTITUTE OF TECHNOLOGY BOMBAY
Powai, Mumbai - 400076
May 2021

Declaration

I declare that this written submission represents my ideas in my own words and where other ideas or words have been included, I have adequately cited and referenced the original sources. I also declare that I have adhered to all principles of academic honesty and integrity and have not misrepresented or fabricated or falsified any idea/data/fact/source in my submission. I understand that any violation of the above will be cause for disciplinary action by the Institute or from whom proper permission has not been taken when needed.

Acknowledgement

I would like to express my deep gratitude to Prof. Sachin Patkar for his kind guidance all through. I also extend thanks to Prof. Virendra Singh for their help in concept clearing. I would also like to thank Mitul Tyagi for their constant help and support.

Gourav Tilwankar

Harsh Paryani

Tanuj Kumar

Deepak Chand

Table of Contents

List of Figures

List of Tables

1. Introduction

Motivation

Motivation behind choosing this project was to understand the underlying concepts of designing a processor, intricacies and complications faced by a designer in this process and solving those intelligently. Also, Implementation of this processor using a Hardware Description Language was an exciting task and kept us focussed.

Objective

The aim of this project is to design a RISC Based 8085 microprocessor with its Instruction Set Architecture (ISA) as a subset of original Intel 8085 Instruction Set. Here, we have tried to design various types of implementation of 8085 ISA viz-a-viz Single Cycle, Multi-Cycle and Pipelined Version. After that, a comparative study is done taking into consideration various aspects on which a processor can be measured.

2. Tools Used

The following tools are used while designing this project:

a) Quartus Prime Lite version 18.1
b) ModelSim Altera
c) De0 Nano Board Field Programmable Gate Array (FPGA)
d) Python 3.9.4

Quartus Prime Lite 18.1

The design of this project is done in Quartus Prime Lite 18.1 version which is a tool used in Hardware Design using Hardware Description Languages such as VHSIC Hardware Description Language (VHDL), Verilog, etc. This project is designed and coded entirely in Verilog language.

ModelSim Altera

The design needs to be simulated on software through which its working can be tested. Hence, the software used for that part is ModelSim Altera. The design is checked on Register Transfer Level (RTL) Simulation and GATE Level Simulation.

De0 Nano Board FPGA

The whole design is tested on an FPGA (De0 Nano Board). The output of few program flows is tested exclusively and checked with that of the simulation part and verified.

Python 3.9.4

Here, Python is used as a medium to convert Assembly Level Language Instructions to Machine Level Instructions and then to be feeded to the designed processor for testing of program flows.

3. Implementation

3.1. Instruction Set Architecture

The Instruction Format is as follows:

Immediate data Instruction format (I)

Opcode		Operand 1/(Empty)		Data (7- bit)	
15	11	10	8	7	0

Single Operand Instruction format (S)

Opcode		Operand 1		Function bit		(Empty)	
15	11	10	8	7	3	2	0

Branch Instruction Format (B)

Opcode		(Empty)		Address (8-bit)	
15	11	10	8	7	0

Double operand instruction format (D)

Opcode		Operand 1		Operand 2	
15	11	10	8	7	0

Implicit Instruction format (Im)

Opcode		(Empty)		Function bit		(Empty)	
15	11	10	8	7	3	2	0

Fig 1: Different Types of Instruction Format

The chosen ISA and its description taken from original 8085 ISA is as follows:

Table 1: Instruction Mnemonics and Description

Mnemonic	Instruction Format	Assembly	Action/Description	Flags
ADD	S	ADD ra	Add content of register ra with content of accumulator.	Z,C
ADC	S	ADC ra	Add the content of register ra with content of accumulator and carry bit.	Z,C
SUB	S	SUB ra	Subtract content of register ra from content of accumulator.	Z,C
SBB	S	SBB ra	Subtract content of register ra and carry bit from content of Accumulator.	Z,C
ANA	S	ANA ra	AND the content of register ra and accumulator.	Z
ORA	S	ORA ra	OR the content of register ra and accumulator.	Z
XRA	S	XRA ra	XOR the content of register ra and accumulator.	Z
CMP	S	CMP ra	Complement the content of register ra.	Z
INR	S	INR ra	Increment the content of register ra.	Z
DCR	S	DCR ra	Decrement the content of register ra.	Z
ADDM	S	ADDM [ra]	Add content of memory with address as ra data with content of Accumulator.	Z,C
ADCM	S	ADCM [ra]	Add content of memory with address as ra data and carry bit with content of Accumulator.	Z,C
SUBM	S	SUBM [ra]	Subtract content of memory with address as ra data from content of Accumulator.	Z,C

SBBM	S	SBBM [ra]	Subtract content of memory with address as ra data and carry bit with content of Accumulator.	Z,C
ANAM	S	ANAM [ra]	AND content of memory with address as ra data with content of Accumulator.	Z
ORAM	S	ORAM [ra]	OR content of memory with address as ra data with content of Accumulator.	Z
XRAM	S	XRAM [ra]	XOR content of memory with address as ra data with content of Accumulator.	Z
CMPM	S	CMPM [ra]	Complement content of memory with address as ra data.	Z
INRM	S	INRM [ra]	Increment content of memory with address as ra data.	Z
DCRM	S	DCRM [ra]	Decrement content of memory with address as ra data.	Z
CMA	Im	CMA	Complement content of accumulator	NONE
CMC	Im	CMC	Complement carry bit.	C
STC	Im	STC	Set carry bit.	C
RET	Im	RET	Return. (Return address is stored in register file address “110”)	NONE
HLT	Im	HLT	Halt	NONE
NOP	Im	NOP	No operation	NONE
ADI	I	ADI Imm	Add content of accumulator to 8 bit immediate data.	Z,C
ACI	I	ACI Imm	Add content of accumulator and carry bit to 8 bit immediate data.	Z,C
SUI	I	SUI Imm	Subtract 8 bit immediate data from content of Accumulator.	Z,C

SBI	I	SBI Imm	Subtract 8 bit immediate data and carry bit from the content of the accumulator.	Z,C
ANI	I	ANI Imm	AND immediate data with content of accumulator.	Z
ORI	I	ORI Imm	OR immediate data with content of accumulator.	Z
XRI	I	XRI Imm	XOR immediate data with content of accumulator.	Z
CPI	I	CPI Imm	Complement immediate data and store in accumulator.	Z
LDA	I	LDA [Imm]	Load accumulator with data present in memory at 8 bit Immediate data address	NONE
STA	I	STA [Imm]	Store accumulator at memory address given by 8 bit Immediate data	NONE
JMP	B	JMP [Imm]	Jump to instruction location PC+1+Immediate data.	NONE
JNZ	B	JNZ [Imm]	If zero bit is reset Jump to instruction location PC+1+Immediate data.	NONE
JNC	B	JNC [Imm]	If the carry bit is reset, Jump to instruction location PC+1+Immediate data.	NONE
CALL	B	CALL [Imm]	Store PC+1 in register file address “110” and jump to instruction location PC+1+Immediate data.	NONE
MVIR	I	MVIR ra Imm	Store Immediate data in register ra.	NONE
MVIM	I	MVIM ra Imm	Store immediate data in memory location given by register ra.	NONE
MOVRR	D	MOVRR ra rb	Move content of register rb to register ra.	NONE
MOVRM	D	MOVRM ra [rb]	Move content of memory location given by register rb to register ra.	NONE

MOVMR	D	MOVMR [ra] rb	Move content of register rb to memory location given by register ra.	NONE
MOVRA	S	MOVRA ra	Move content of accumulator to register ra.	NONE
MOVAR	S	MOVAR ra	Move content of register ra to accumulator.	NONE
MOVAM	S	MOVAM [ra]	Move content of memory location given by register ra to accumulator.	NONE
MOVMA	S	MOVMA [ra]	Move content of accumulator to memory location given by register ra.	NONE
MVIA	I	MVIA Imm	Store immediate data in the accumulator.	NONE

Note- The programmer needs to keep in mind that in the two operand instructions like MOVRR Ra Rb, in single cycle and pipeline implementations, the first operand is destination and second operand is source while in Multicycle implementation the first operand is source and second operand is the destination**.**

3.2. Multicycle Implementation

The Resources used in this type of Implementation are as follows:

1. ALU – 8-bit
2. Flags : ZERO and CARRY
3. Program Counter (PC)- 8bit
4. Register Files – 7 Registers of 8-bit each
5. Temporary Registers – 2 Registers of 8-bit each
6. ALU Controller
7. 256*16 bit Memory(common for instruction and data)
8. Accumulator – 8 bit
9. Instruction Register- 16 bit
10. MUXs according to need for control signal selection
11. Control path/ Controller
12. Clock being used is of 100 MHz (for Simulation) and 50 MHz (on FPGA).

State Diagram for MultiCycle Implementation is as follows:

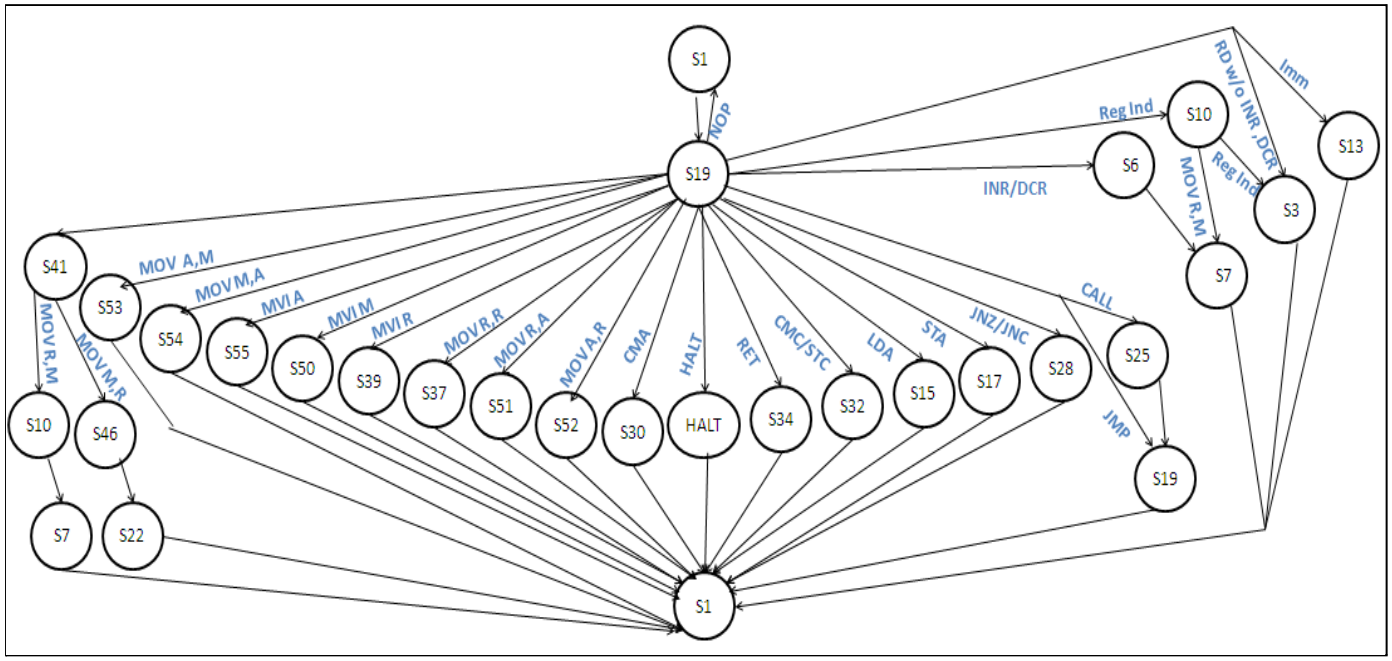

Fig 2: State Transition Diagram for Multi-Cycle

Link for clear figure is :
https://drive.google.com/file/d/1KNao5LbQuHyYHimCThk242KCxn7iFbvy/view?usp=sharing

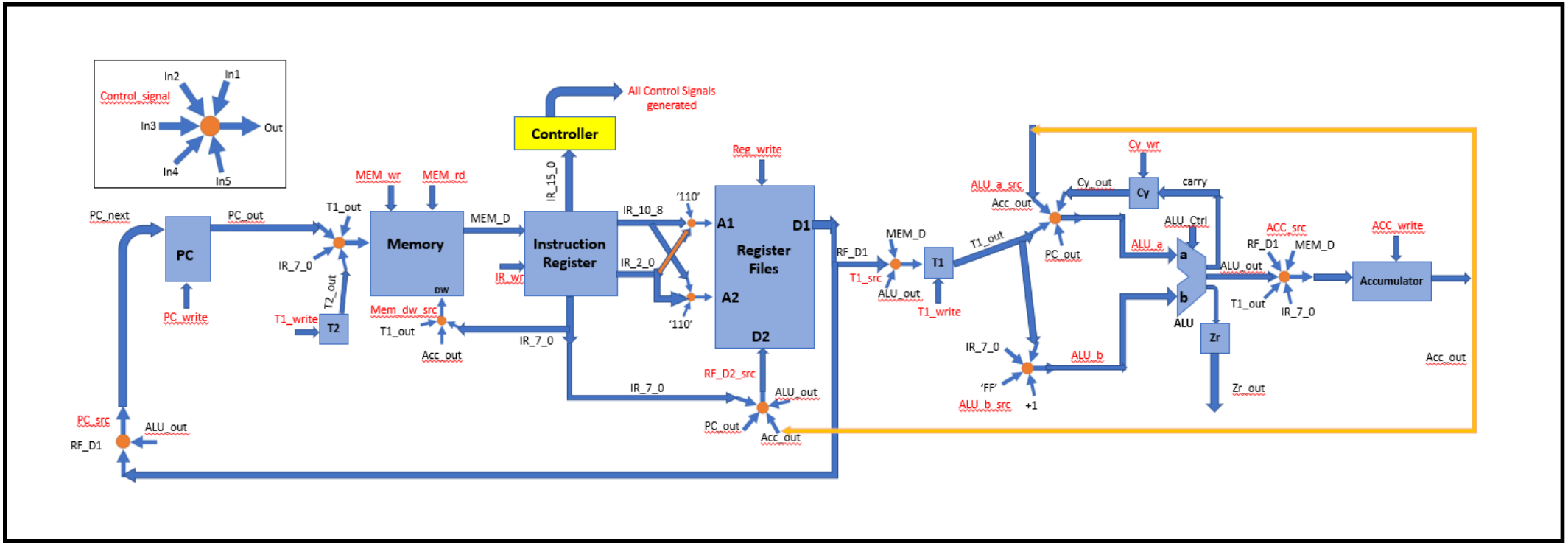

Fig 3: Datapath for Multicycle Implementation

Link for clear figure is: https://drive.google.com/file/d/1FTLwgxSmgkjLq1H-DyOXbdlaRipPIvSA/view?usp=sharing

3.3. Single Cycle Implementation

The Resources used in this type of Implementation are as follows:

1. Program Counter (PC) - 8 bit
2. 3 ALU's – 8 bit each
3. Flags : ZERO and CARRY
4. Register Files – 7 Registers of 8 bit
5. Accumulator – 8 bit
6. Data Memory
7. ALU Controller
8. 256*16 bit Instruction Memory
9. 256*16 bit Data Memory
10. Control path
11. MUXs according to need for control signal selection
12. Clock being used is of 25 MHz (for Simulation) and 12.5 MHz (on FPGA)

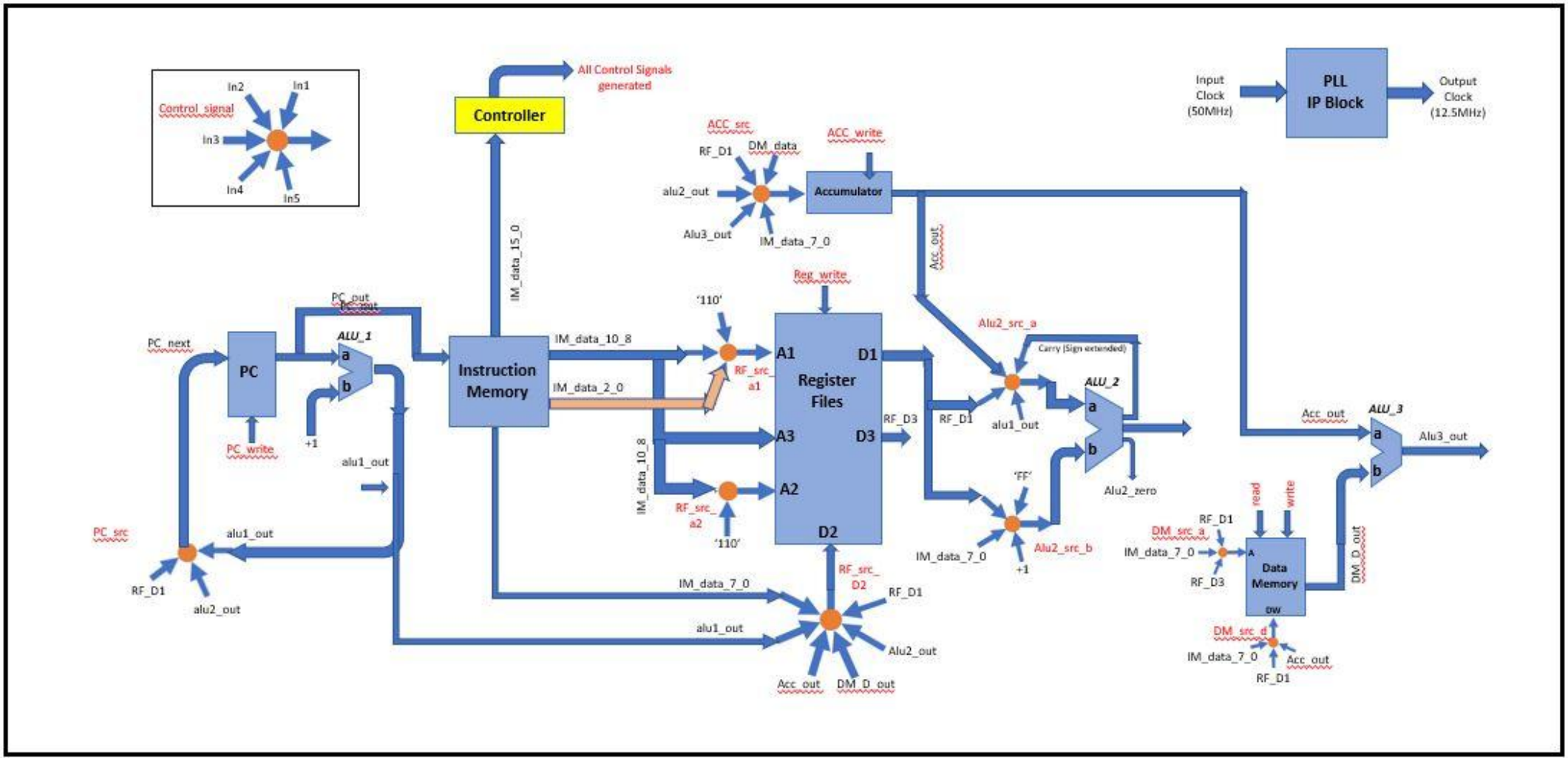

Fig 4: Datapath for Single Cycle Implementation

Link for clear figure is:
https://drive.google.com/file/d/1gRysgsxuSePGnDLhBYdnhcr9EPlWqa_Z/view?usp=sharing

3.4. Pipelined Version Implementation

The Resources used in this type of Implementation are as follows:

1. ALU – 8 bit
2. Flags : ZERO and CARRY
3. Program Counter (PC) -8 bit
4. Register Files – 7 Registers of 8 bit
5. Temporary Registers – 3 Registers of 8 bit each
6. Accumulator – 8 bit
7. 256*16 Data Memory
8. ALU Controller
9. 256*16 Instruction Memory
10. MUXs according to need for control signal selection
11. Branch prediction Look-up Table
12. Hazard control block
13. Control path
14. 4 Stage Transition Registers.
15. Clock being used is of 100 MHz (for Simulation) and 50 MHz (on FPGA)

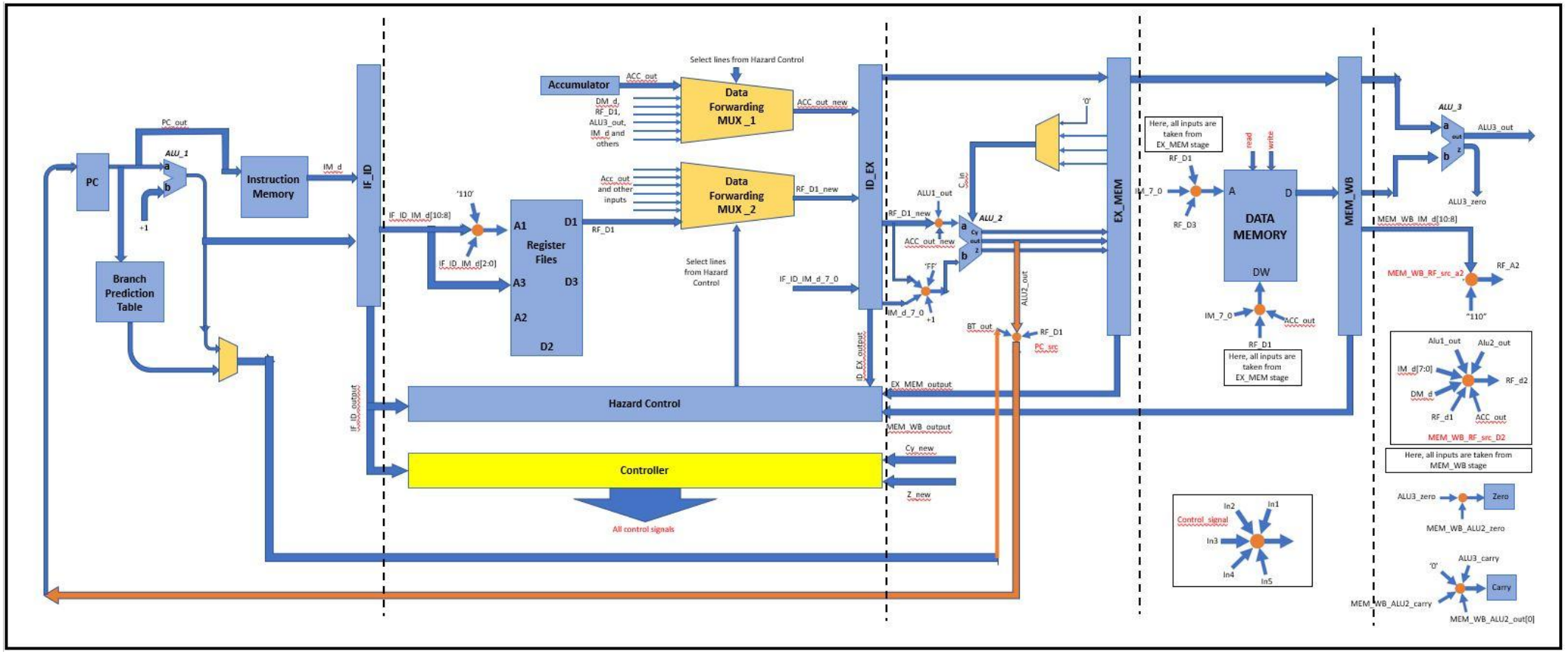

Fig 4: Datapath for Pipeline Implementation

Link for clear figure is:
https://drive.google.com/file/d/1GtYiaFhRnhZbD7tCAWAvPSjE26APpd5S/view?usp=sharing

3.5. Usage Instructions for this Processor

The machine can only understand 0's and 1's, i.e., binary language. Hence if we write any operation E.g MOVRR B C, it is not understandable by the machine. We need to write the binary equivalent of this operation by looking into the code or an associated document everytime we want to implement a program. This can be quite hectic and tiresome in regular use.

To remove this sort of overwork, we have created a python file which can convert the program written in Assembly language directly into the machine language equivalent which is done actually by our 8085 machine. The programmer now only needs to follow these steps:

1) Open a .txt file in the same folder in which ***test_gen_8085.py*** is stored and write the code he/she wants to implement. He/She needs to take care that there are no unnecessary spaces or extra lines between or at the end of the code. If there is some error in writing the assembly code it would be represented by "**F800**" in the converted machine code. Check whether the instruction written is in correct format or not. If there is, please check your code and re-run the python code.

2) Open a powershell window or terminal in the same folder and write

   a. *python3 test_gen_8085.py <name of inputfile.txt> <name of outputfile.txt>*

   b. Here the test_gen_8085.py is the python file used for the conversion of Assembly code to Machine equivalent program.

3) A file named outputfile.txt (the name of the output file you gave in powershell/terminal window) will be created in the same folder. Now when the programmer compiles the verilog program, the machine equivalent code from outputfile.txt would itself be stored in the Instruction memory of the program and one can just simulate it in Modelsim to check the results.

4. Observations

4.1. Performance Evaluation

We have implemented 5 test codes to check the performances of the three implementations. These 5 test codes are:

4.1.1 Factorial of a number

The program calculates the factorial of any number. We have checked it for 5.

4.1.2 Exponential Calculation

The program calculates a^N of a number where a and N can be decided by the programmer. We have checked it for 5^3.

4.1.3 Factorial using Call instruction (function implementation as compared to C Language)

The program calculates factorial number using call instruction. By using the call instruction we are calculation factorial in a manner similar to what a C program would do using a function.

4.1.4 Sorting of numbers in an array

The program sorts the numbers stored in data memory addresses 11 to 18 in single cycle/pipeline implementation. For the Multicycle implementations the numbers are stored from 128-135 of the memory. The program uses the Selection Sort algorithm to sort these numbers in descending order.

4.1.5 Smallest number of the array

8 numbers are stored in data memory addresses 11 to 18 in single cycle/pipeline implementation. For the Multicycle implementations the numbers are stored from 128-135 of the memory. Program finds the smallest number out of these.

4.2. Comparison

Table 2: Performance comparison of Different Implementations

Sr. No.	Implementation	Resources Used		Max Frequency allowed (MHz)
		Logic Elements	Registers	
1.	Single Cycle	3472	2130	34.98
2.	Multi cycle	3914	2476	103.49
3.	Pipeline	4216	2479	104.34

Table 3: Time Consumed for Total Program Execution (nanoseconds)

Implementation Type	Single Cycle	Multi - Cycle	Pipelined Version
Program used			
Exponential Calculation (5^3)	3040	2580	950
Sorting of Numbers in array	13560	12780	4180
Finding Smallest Number in Array	1680	1500	650
Factorial of a Number using CALL Instruction	3320	2750	1060
Factorial	2160	1780	710

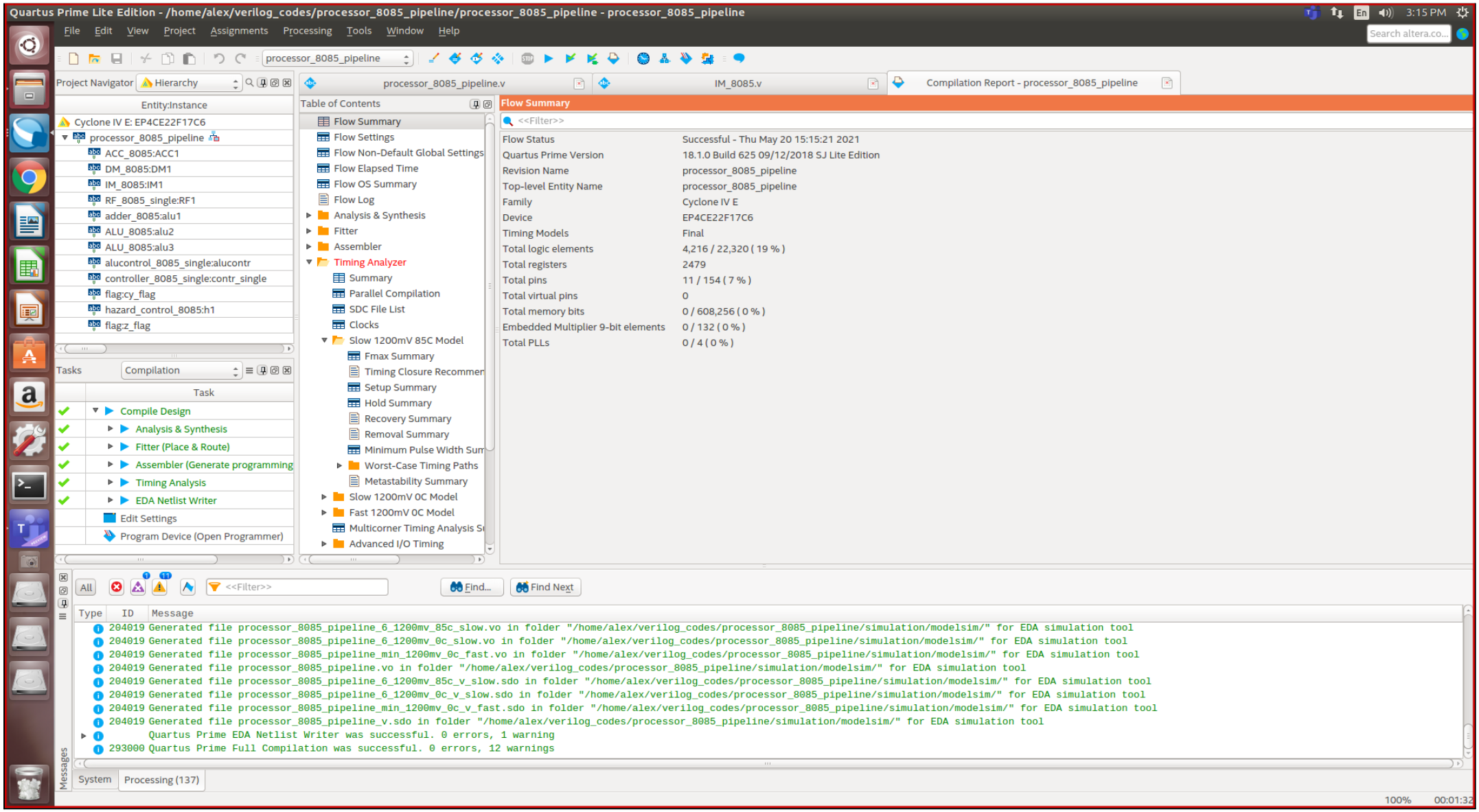

Fig 6: Compilation Report for Pipelined Processor showing Resource Usage

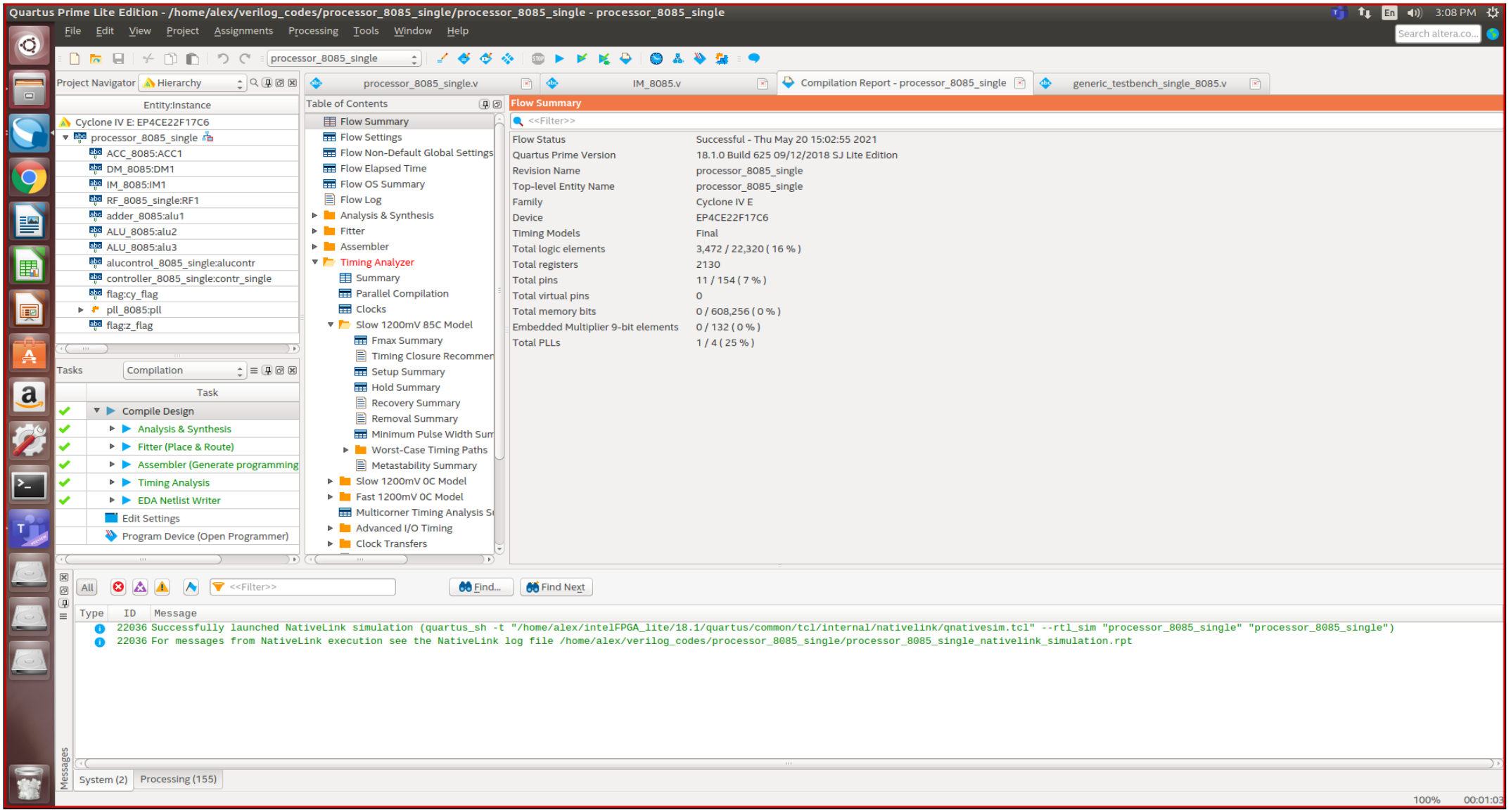

Fig 7: Compilation Report for Single Cycle Processor showing Resource Usage

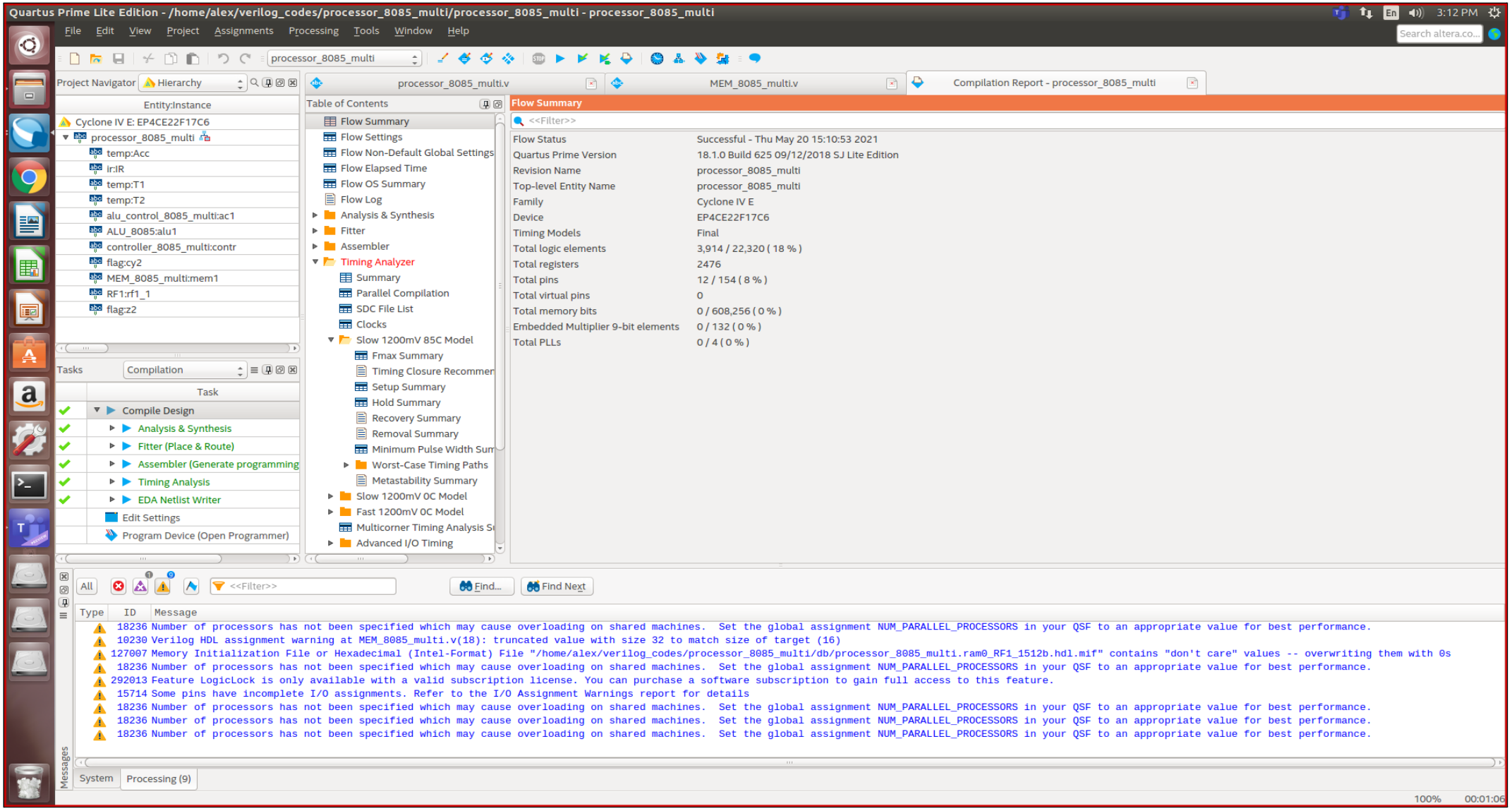

Fig 8: Compilation Report for Multi Cycle Processor showing Resource Usage

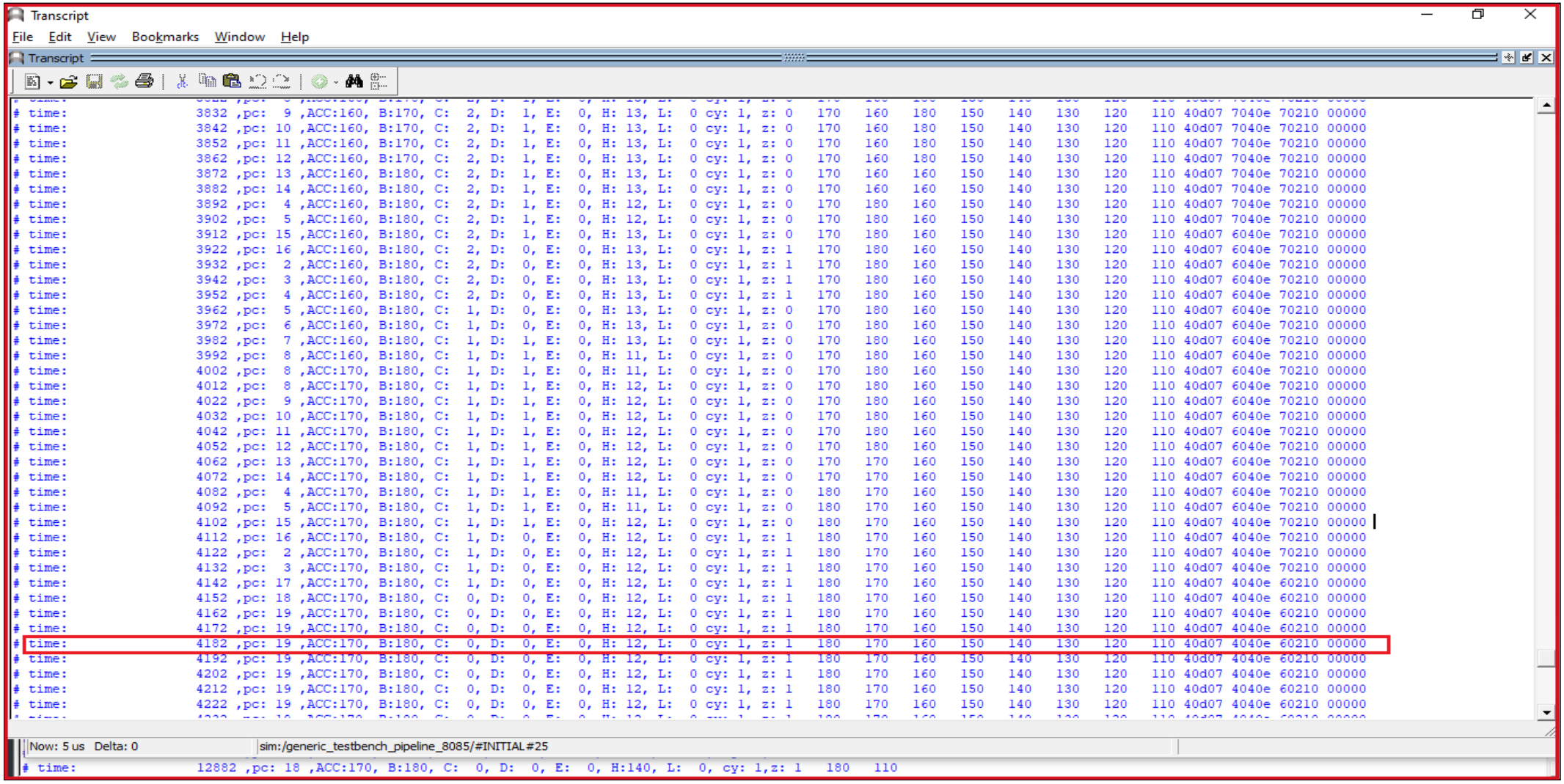

Fig 9: Modelsim output for Pipelined Processor showing Timing of Program for Sorting of Numbers in Array

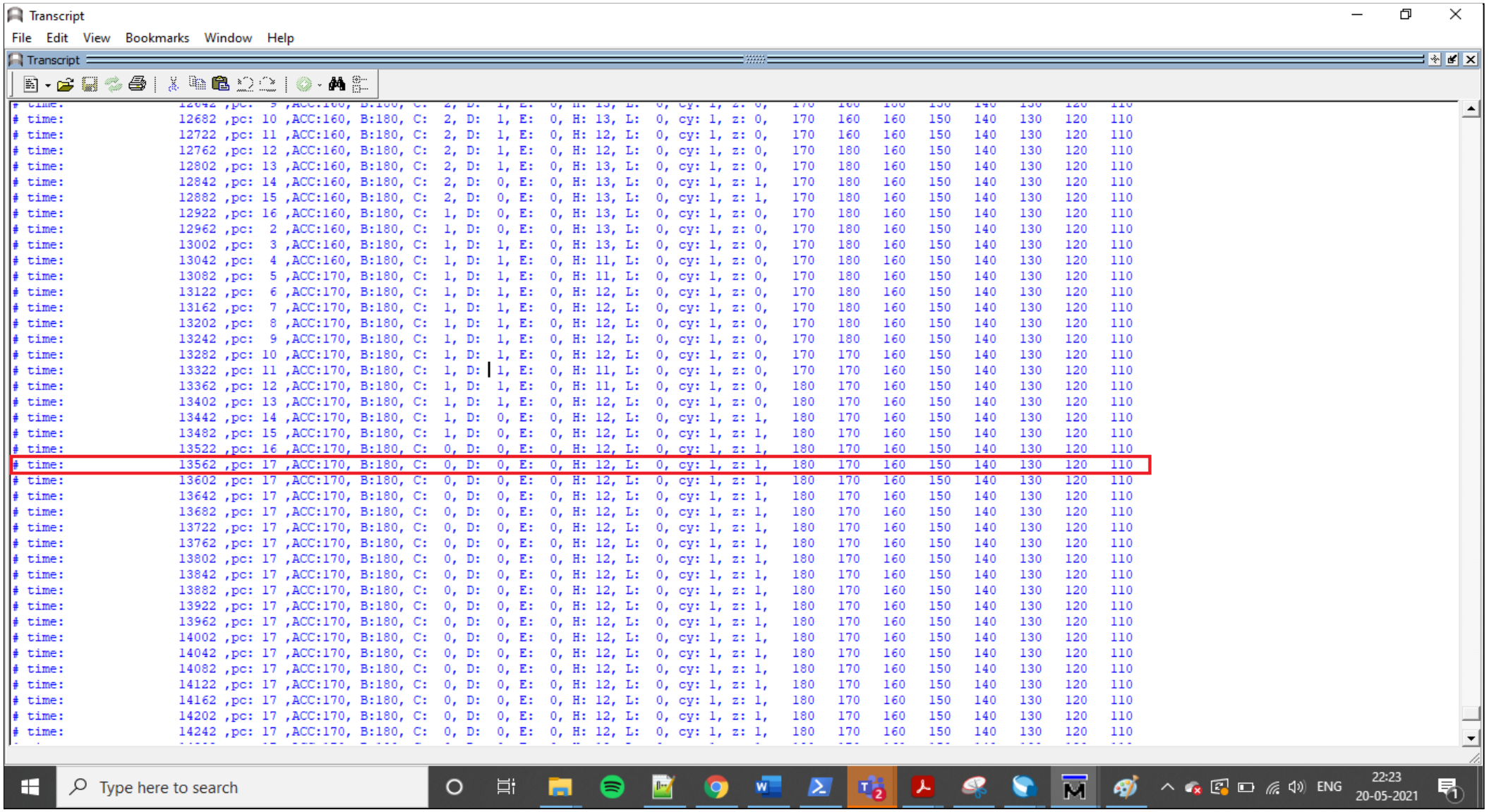

Fig 10: Modelsim output for Single Cycle Processor showing Timing of Program for Sorting of Numbers in Array

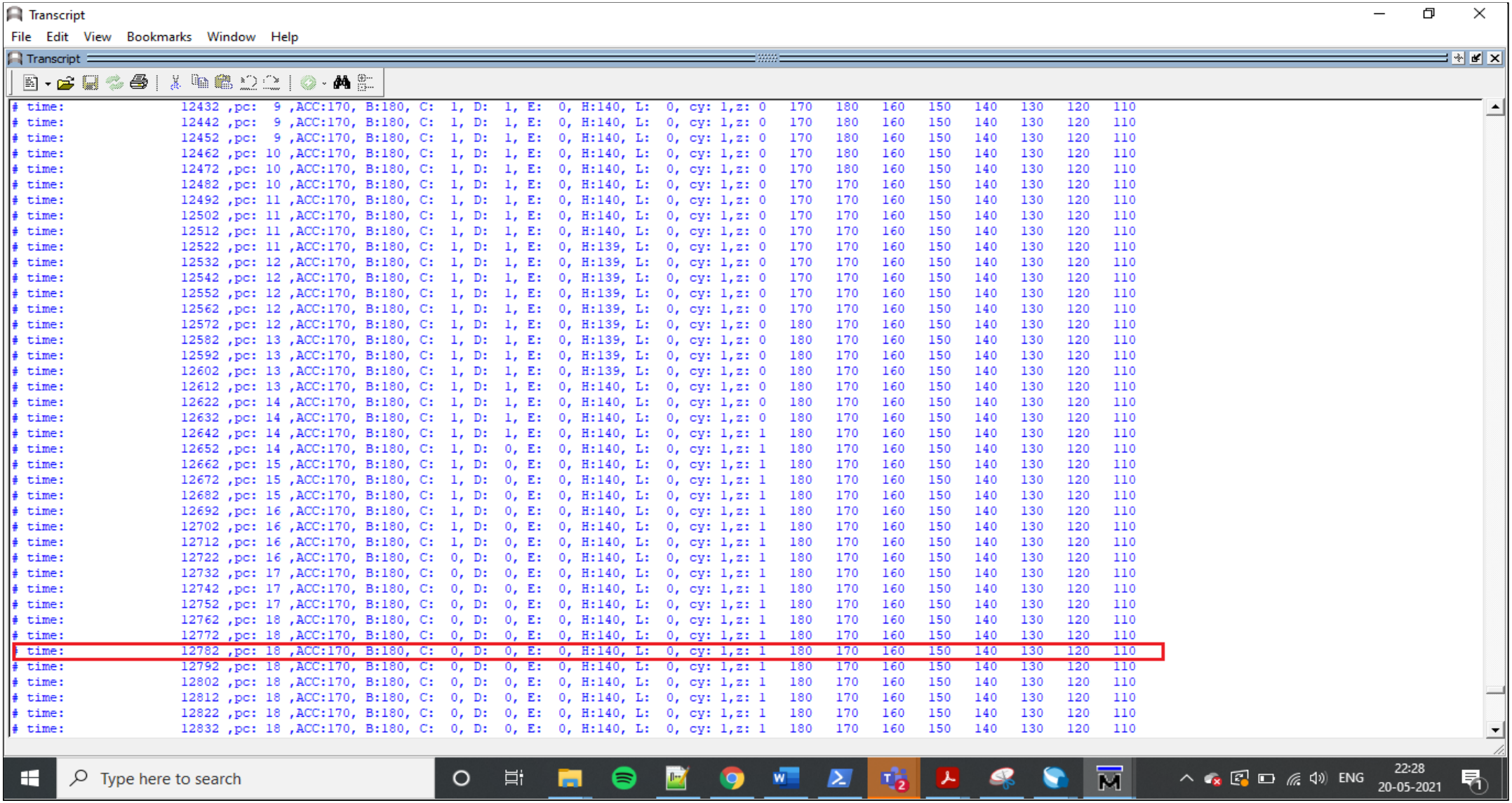

Fig 11: Modelsim output for Multi-CycleProcessor showing Timing of Program for Sorting of Numbers in Array

5. Conclusion

The testing of various kinds of programs like having just one set of instructions in a program to programs including heavy usage of all kinds of instructions like calculating factorial of a number and sorting of memory elements in descending order is done.

Performance Analysis shown above clearly states that for the particular Clock Frequency chosen, the Pipelined Version of our Project gives the best results in all types of programs.

Also, according to resource usage of different implementations shows Pipelined version is using most number of resources (Logic Elements and Registers), while giving the best Maximum Frequency that can be given to the implementation and giving almost running 3 times faster that Single Cycle and around 2.5 times faster than Multi-Cycle Implementation.

6. Future Work

All kinds of implementations of processors that take a single instruction at a time for processing is carried out in this project. A Future implementation of this Project including SuperScalar Version can be carried out which can take multiple instructions for processing. So that the program can be executed even earlier.

7. References

- https://www.javatpoint.com/instruction-set-of-8085
- https://www.slideshare.net/anupamkumarpandit/list-of-8085-programs
- Tredennick N., *Microprocessor Logic Design-The Flowchart Method*

8. URL Section

Link for Project Codes, report and images:
https://drive.google.com/drive/folders/1oAR_hNOS_0Kr9wiqTpJIjWNwWL-sGuRW?usp=sharing
Link for Code Walkthrough and usage:
https://drive.google.com/file/d/1IP985SiRRBzsUL_YH8k1AwJDxVvFakYs/view?usp=sharing
Link for FPGA Demo:
https://drive.google.com/file/d/1cnXNf7UlMSAL7I0TXZEDH0lH0tF-iYXZ/view?usp=sharing

9. Contribution towards Project

Table 4: Contribution Table

Task	Done By
Instruction Set Definition	Harsh and Gourav
Multi-Cycle Implementation (Datapath, Control Path and testing)	Tanuj Deepak, Harsh and Gourav
Single Cycle - Datapath	Harsh and Gourav
Single Cycle - Control Path	Tanuj and Deepak
Single Cycle Testing	Tanuj Deepak, Harsh and Gourav
Pipelined Version - Datapath, Hazard Control and Branch Predictor	Harsh
Pipelined Version - Control Path	Tanuj and Deepak
Pipelined Version - Testing	Gourav and Harsh
Testbench Generation	Gourav
Datapath figures/Images	Gourav
Report	Gourav, Tanuj, Deepak and Harsh