EDUCATIONAL PROCESSOR

By

NIK ADLI HAKIMI BIN NIK MOHAMAD SHUKRI

DISSERTATION

Submitted to the Electrical & Electronics Engineering Programme
in Partial Fulfilment of the Requirements
for the Degree
Bachelor of Engineering (Hons)
(Electrical & Electronics Engineering)

Universiti Teknologi PETRONAS
Bandar Seri Iskandar
31750 Tronoh
Perak Darul Ridzuan

CERTIFICATION OF APPROVAL

EDUCATIONAL PROCESSOR

by

NIK ADLI HAKIMI BIN NIK MOHAMAD SHUKRI

A project dissertation submitted to the
Electrical & Electronics Engineering Programme
Universiti Teknologi PETRONAS
in partial fulfilment of the requirement for the
Bachelor of Engineering (Hons)
(Electrical & Electronics Engineering)

Approved:

Patrick Sebastian
Project Supervisor

UNIVERSITI TEKNOLOGI PETRONAS
TRONOH, PERAK

September 2011

CERTIFICATION OF ORIGINALITY

This is to certify that I am responsible for the work submitted in this project, that the original work is my own except as specified in the references and acknowledgements, and that the original work contained herein have not been undertaken or done by unspecified sources or persons.

Nik Adli Hakimi Bin Nik Mohamad Shukri

ABSTRACT

This report discusses about the overview of the chosen project, which is an Educational Processor (EduCPU). The objective of this project is to develop a simple processor using TTL logic gates and also to develop simulation software for educational purpose. The software is responsible for sending instruction codes to the simple processor through serial communication in order to execute the instruction. The software written is also capable of simulating the behaviour of the simple processor. This educational processor would be used as a learning tool in Computer System Architecture course in in Universiti Teknologi PETRONAS (UTP) to assist students in understanding about computer system architecture. In order to complete this project, the scope of study basically will cover the computer system architecture and details about Central Processing Unit (CPU). The instruction format and CPU data path design both are based on MIPS architecture processor. The methodologies that are involved in this project are design and validation phase, constructing the hardware, and programming the user interface to interact with the educational processor.

ACKNOWLEDGEMENTS

Firstly, I give my utmost gratitude to ALLAH the Almighty for his uncountable graces upon me and for the successful completion of this project in due course of time.

I would like to express the appreciation to my supervisor, Mr. Patrick Sebastian, Lecturer of Electrical & Electronics Department, UTP. The supervision and continuous support that he gave truly helped me throughout completing this project. He provided lots of guide, sample codes, and teaching me concepts in order to successfully complete this project. He also helped me in correcting various documents of mine with attention and care.

Lastly, great appreciation to my friends, who always helped me and giving me support when I needed it. Not to forget my appreciation to all UTP lecturers, students, staff, friends and to all whose their names are not mentioned here but they provided help directly or indirectly in completing my project.

TABLE OF CONTENTS

ABSTRACT iv

ACKNOWLEDGEMENTS v

TABLE OF CONTENTS vi

LIST OF FIGURES ix

LIST OF TABLES xi

LIST OF ABBREVIATIONS xii

CHAPTER 1 INTRODUCTION 1

1.1 Background of Study 1

1.2 Problem Statement 1

1.2.1 Problem Identification 1

1.2.2 Significance of Project 2

1.3 Objective and Scope of the Project 2

1.3.1 Objectives 2

1.3.2 Scope of the Project 3

1.4 Relevancy of Project 3

1.5 Feasibility of Project 3

Chapter 2 LITERATURE REVIEW 4

2.1 Introduction to Processors 4

2.2 Instruction Set Architecture 4

2.3 Introduction to MIPS Architecture 5

2.4 CPU Functional Units 6

2.4.1 Program Counter (PC) 7

2.4.2 Instructions Memory 7

2.4.3 Instruction Register 7

2.4.4 Register File 7
2.4.5 Random Access Memory (RAM) 7
2.4.6 Control Logic 8
2.4.7 Arithmetic and Logic Unit (ALU) 8
2.4.8 Address Bus and Data Bus 8
Chapter 3 METHODOLOGY 9
3.1 Project Flowchart 9
3.2 Research Methodology 10
3.3 Instruction Set Design 10
3.4 Tools Required 12
3.5 Instructions List 13
3.6 Datapath 14
3.7 Graphical User Interface 14
3.8 Project Duration 14
Chapter 4 RESULTS AND DISCUSSION 15
4.1 Graphical User Interface 15
4.2 Compiling 20
4.2.1 R-Type 20
4.2.2 I-Type 20
4.2.3 J-Type 21
4.3 Simulation: Test Code 1 – Adding values 21
4.4 Simulation: Test Code 2 – Multiplying values 25
4.5 Transmitting machine code through serial communication 30
4.6 Hardware 31
4.7 Discussion 33
Chapter 5 CONCLUSION & RECOMMENDATIONS 35
5.1 Conclusion 35

5.2 Recommendations 35
APPENDIX A – PROJECT GANTT CHART 37
APPENDIX B – DATAPATH DESIGN 39
APPENDIX C – CIRCUIT DESIGN 40
APPENDIX E – PERL SOURCE CODE 44
APPENDIX F – VISUAL BASIC 2010 SOURCE CODE 47

LIST OF FIGURES

Figure 1: MIPS R-Type instruction format 5
Figure 2: MIPS I-Type instruction format 5
Figure 3: MIPS J-Type instruction format 6
Figure 4: Project Flowchart 9
Figure 5: Instruction Format Design 11
Figure 6: R-Type instruction format 11
Figure 7: I-Type instruction format 11
Figure 8: J-Type instruction format 11
Figure 9: EduCPU Main View 15
Figure 10: Example of code with errors 16
Figure 11: Code without any syntax error 17
Figure 12: EduCPU Datapath View 17
Figure 13: Register View 18
Figure 14: Memory View 18
Figure 15: Help Window 19
Figure 16: Test Code 1, Line 1 Datapath View 22
Figure 17: Test Code 1, Line 1 Register View 22
Figure 18: Test Code 1, Line 3 Datapath View 23
Figure 19: Test Code 1, Line 3 Register View 23
Figure 20: Test Code 1, Line 4 Memory View 24
Figure 21: Reg 3 = Reg 1 + Reg 3 25
Figure 22: Registers view after line 4 is transmitted 26
Figure 23: Registers view after line 5 is transmitted 26
Figure 24: Datapath view after transmitting line 6 27
Figure 25: Main view after transmitting line 4 for second time 27
Figure 26: Datapath view after transmitting line 4 for second time 28
Figure 27: Registers view after transmitting line 4 for second time 28
Figure 28: Datapath view after transmitting the last line of the code 29
Figure 29: Registers view after transmitting the last line of the code 29
Figure 30: Main view after transmitting the last line of the code 30
Figure 31: Connection between PC and hardware 30

Figure 32: Choosing the correct serial port 31
Figure 33: Hardware of Educational Processor 32

LIST OF ABBREVIATIONS

RISC	Reduced Instruction Set Computing
CISC	Complex Instruction Set Computing
MIPS	Microprocessor without Interlocked Pipeline Stages
CPU	Central Processing Unit
RAM	Random Access Memory
EPROM	Erasable Programmable Read-only memory
PC	Program Counter
CSA	Computer System Architecture
TTL	Transistor-transistor Logic
OPCODE	Operation Code

CHAPTER 1
INTRODUCTION

This chapter discusses about the introduction to this project. It covers the background of study which discusses the background knowledge involved in this project. The problem statement and the reasons that lead to the implementation of this project are also discussed in this chapter.

1.1 Background of Study

This project is aimed to develop a simple educational processor which would be used as a teaching material in Computer System Architecture class. The main objective of this project is to provide an opportunity for the students taking this course to understand and examine how a processor executes an instruction. Students will be able to interactively interact with the basic of the processor to enhance students' learning environment.

The knowledge required in this project is the knowledge of digital electronics and also knowledge about computer system architecture. This project also requires the knowledge in microcontroller since this educational processor would be interfaced to a computer using a microcontroller via serial communication. In order to write the program that would be used to interface the educational processor and computer, knowledge about programming using C, Perl, and Visual Basic is also needed.

1.2 Problem Statement

1.2.1 Problem Identification

The processor is one of the most important parts of a computer system. The development of the processor has evolved over the years. In 1945, a mathematician John Von Neumann outlined the design of a stored-program computer which became the primary design of most modern Central Processing Units (CPU) [1]. Most of the processor designs now are very sophisticated and complex compared to its earlier development stage. This makes the learning process of how processors actually work becomes increasingly difficult.

The Electrical and Electronics Engineering students in Universiti Teknologi PETRONAS especially those taking Computer Systems as their major have the chance to learn about computer system through Computer System Architecture course. The course exposes the students to lectures and also lab assignments in order for the students to understand the basics of computer system architectures, including on how processors work.

The course does not focus on any specific computer architecture, but instead exposes the students to the general processor designs with MIPS architecture processor briefly explained. The course itself is also quite theoretical which makes it harder for the students to fully understand the concepts of processors.

1.2.2 Significance of Project

This project would give an opportunity to the Computer System Architecture students to explore and examine at the gate level about a MIPS-based architecture processor datapath. The students would be able to observe exactly what happens at each stage in the processors and how each logic device interact with each other in order to complete a CPU instruction.

1.3 Objective and Scope of the Project

1.3.1 Objectives

The main objective of this project is to develop a simple MIPS based architecture processor as learning and teaching tool in Computer System Architecture course.

The sub objectives of the project are listed as the following:

- To help students understand more about how a processor works.

- To construct the PCB boards and validate the prototype.
- To develop a software with a graphic user interface in order to give commands to the designed processor.

1.3.2 Scope of the Project

This project will start with literature reviews related to processors with MIPS architecture to fully understand how processors with MIPS architecture work. After that, the simple processor will be designed in design phase before actually implementing the design on real hardware. The software will also be designed in the design phase in order to let the processor communicate with a connected computer. Then, the prototype will be developed where the data path hardware is implemented using TTL logic gates designed during design phase. Further testing will be carried out to make sure the processor works by interfacing the processor with a computer.

1.4 Relevancy of Project

This educational processor will follow the format of MIPS architecture commands that is included in Computer System Architecture course syllabus. Instead of learning only in theory about how MIPS processors work, students taking Computer System Architecture course will also have the opportunity to clearly see how MIPS processors work. The educational processor will be combined with graphical user interface software. This project will significantly improve the students' understanding about processors, especially MIPS processors.

1.5 Feasibility of Project

The whole project will be done in two semesters. This includes three main areas which are research, development, and also improvement of the design. The software development tools (Microsoft Visual Studio 2010, Perl, MPLAB IDE, and PICKit) are available. The components needed for hardware implementation such as TTL gates and microprocessors are also readily available in the lab. Based on the description above, it is very clear that this project is feasible to be completed within the time frame.

CHAPTER 2
LITERATURE REVIEW

This chapter discusses about the theories and paperwork reviews related to this project. Besides that, details on the educational processor's architecture and data path design would also be discussed in this chapter.

2.1 Introduction to Processors

The processor or CPU is the portion of a computer system that carries out the instructions of a computer program, and is the primary element carrying out the functions of the computer or other processing device. The CPU carries out each instruction of the program in sequence, to perform the basic arithmetical, logical, and input/output operations of the system [2]. This term has been in use in the computer industry at least since the early 1960s [1]. The form, design and implementation of CPUs have changed dramatically since the earliest examples, but their fundamental operation remains much the same.

2.2 Instruction Set Architecture

The Instruction Set Architecture is the part of the processor that is visible to the programmer or compiler writer. It is an abstract model of a computer that describes what it does, rather than how it does it (functional definition). So, it can be said that the instruction set architecture and the instructions available in the processor determine the processor capabilities and performance [3]. The ISA also serves as the boundary between software and hardware.

The ISA varies from machine to machine. Instructions are classified by format and the number of operands they take. The three basics instruction types are data movement which copies data from one location to another, data processing which operates on data, and flow control which modifies the order in which instructions are executed. Instruction formats can take zero, one, two or three operands. It depends on how many bits are used to represent the whole instructions.

2.3 Introduction to MIPS Architecture

MIPS architecture is a 32-bit RISC instruction set architecture developed by MIPS Computer System (now known as MIPS Technologies). MIPS architecture is designed for high performance. To allow the user to get maximum performance, the complexity of individual instructions is minimized. This allows the execution of these instructions at significantly higher speeds [5]. MIPS instructions are classified into groups according to their coding formats [4]. These formats are:

- R-Type (register-to register instruction)

  This group contains all instructions that do not require an immediate value, target offset, memory address displacement, or memory address to specify an operand. This includes arithmetic and logic with all operands in registers, shift instructions, and register direct jump instructions.
- I-Type (immediate operand)

  This group includes instructions with an immediate operand, branch instructions, and load and store instructions. In the MIPS architecture, all memory accesses are handled by the main processor, so coprocessor load and store instructions are included in this group.
- J-Type (branch/jump instruction)

  This group consists of the direct jump instructions. These instructions require a memory address to specify their operand.

Figures below describe the format of 32-bit MIPS instruction formats – R-Type, I-Type, and also J-Type instructions.

31 26	25 21	20 16	15 11	10 6	5 0
Opcode (6)	RS (5)	RT (5)	RD (5)	SA (5)	Function (6)

Figure 1: MIPS R-Type instruction format

31 26	25 21	20 16	15 0
Opcode (6)	RS (5)	RT (5)	Immediate (16)

Figure 2: MIPS I-Type instruction format

31 — 26	25 — 0
Opcode (6)	Target Jump Address (26)

Figure 3: MIPS J-Type instruction format

The opcode field is the Operation Code field that indicates the code for each instruction. Each instruction has its own unique opcode. RS is the source or base register. RT acts as a second source register for R-Type instruction, but acts as the destination register for I-Type instruction. RD is the destination register (only present in R-Type instruction). SA (shift amount) is the amount of bits to be shifted. Only certain instructions use this field for execution. Immediate acts as the immediate operand or as address offset, depending on the instruction that is being executed. Target Jump Address is the memory word address to be jumped to [4]. It has 26-bit literal that is concatenated with the 6 most significant bits of the program counter to create 32-bit address.

Since 5 bits are allocated to registers (RS, RT, and RD) field, it follows that the MIPS architecture contains $2^5 = 32$ internal registers that can be accessed by instructions given.

2.4 CPU Functional Units

It is important to understand the relationships between the CPU, the memory, and the program before looking into the details of how CPU works. The program is the list of instructions to be executed by the processor. Examples of programs are the software and applications that are available in our computers today. The memory temporarily stores the list of instruction of the program and also the data of the program during CPU execution. The CPU then reads the list of instructions stored in the memory one-by-one and performs the required execution on the data. Finally, the processed data is stored back into the memory.

2.4.1 Program Counter (PC)

Program counter contains the next instruction address to be executed. Normally, PC will be increased after every instruction executed to point to the next address, except if any flow control instructions is executed which modifies the bits contained in the PC, thus modifies the sequence of the instructions.

2.4.2 Instructions Memory

The instructions memory contains the list of instructions to be executed by the CPU. The CPU fetches the instructions that it needs to execute from the instructions memory.

2.4.3 Instruction Register

Instruction register contains the current instruction that the CPU is executing. It stores the current register temporarily and is connected to various other logic devices such as control logic and register files. After current instruction is completed, the content of this instruction register will be overwritten by the next instruction.

2.4.4 Register File

Register file serves as the general purpose register to store temporary data that is executed by the CPU. Register files are similar to Random Access Memory (RAM), except that it does not have as much capacity as RAM. It is also faster than RAM. This makes the execution of register-register instructions faster.

2.4.5 Random Access Memory (RAM)

RAM is a form of computer data storage. However, it is a volatile memory which means that the stored information is lost if the power is removed. It functions similarly to the register files – to store temporary data. However, it usually has much more capacity than the register file has.

2.4.6 Control Logic

Control logic controls the sequence and the datapath flow of an instruction. When an instruction is executed, it fetches and translates the opcode of the instruction. Then, it will output the control logic signals to the appropriate modules such as register files, ALU, memory, and also multiplexers that handle the data path.

2.4.7 Arithmetic and Logic Unit (ALU)

ALU is the unit that does the arithmetic and logical manipulation of data such as addition, subtraction, logical AND, logical OR, logical XOR, and many more. It is a fundamental building block of the CPU of a computer [6].

2.4.8 Address Bus and Data Bus

Buses are used to simplify the movement of data from point to point in a CPU. It is connected to various logic devices in order to transfer data. In a bused system, only one communication from point to point could happen at any one time. Thus, a careful synchronization needs to be taken care of to make sure that the data is successfully transmitted.

CHAPTER 3
METHODOLOGY

This chapter discusses about how the project would be carried out. It includes the method of research, tools, components, and software involved.

3.1 Project Flowchart

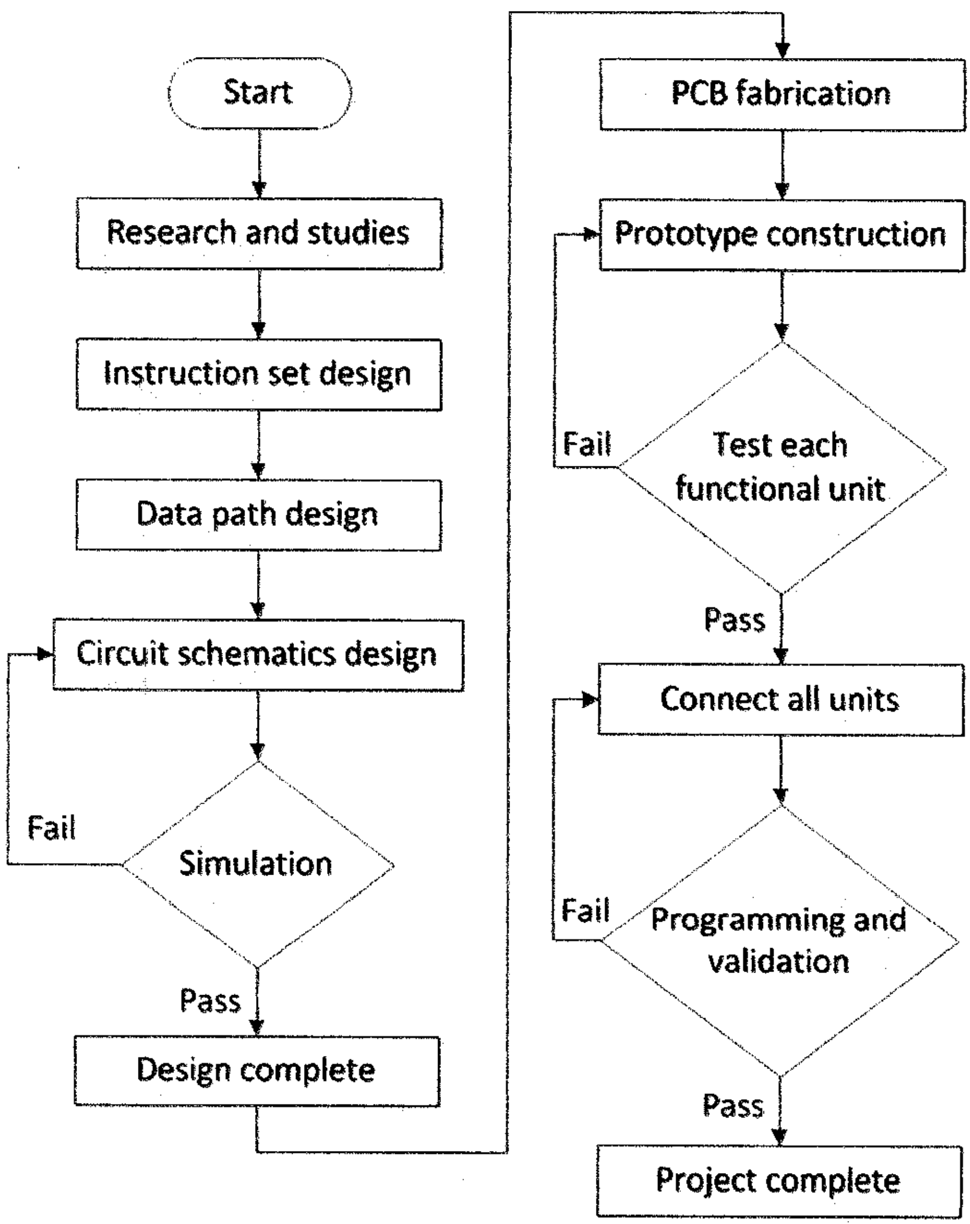


Figure 4: Project Flowchart

3.2 Research Methodology

The theories behind CPU design and CPU architectures, especially MIPS architecture are studied. However, studies regarding CISC and RISC architectures are also done. The research mainly focuses on the decision between these architectures – which one would best educate students.

After decision is made, which is to focus on MIPS architecture, further research regarding MIPS architecture is done. To understand CPU architecture, the knowledge on these theories is important. This includes instruction set architecture, CPU functional units, and CPU data path. All these have been outlined in the literature review chapter.

The sources of research include from books, websites, conference papers, and also journals. The relevancy between the selected sources and the project's objectives is also taken into account to ensure the credibility of this project. The understanding from participation in Computer System Architecture classes and labs has also contributed to the research study.

3.3 Instruction Set Design

In this stage, instruction set architecture is designed. The ISA defines the whole identity of the processor itself. Since the processor is only used for educational purposes, which is to show to the students the concept of how processors work, it would have a very limited instruction set. Thus, the choice of instructions to be included has been made according to research that shows the most commonly used instruction in a program.

The design begins with the format of the instruction set. The instruction format defines the width of the instruction, opcode field, and also operand fields. Figure 5 below illustrates how the instructions format is designed.

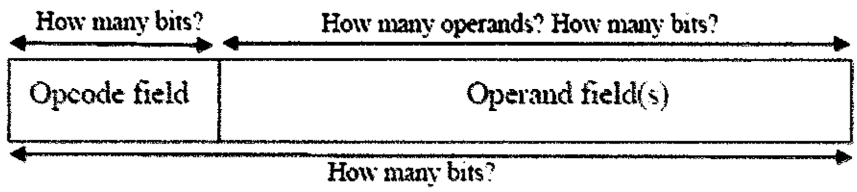


Figure 5: Instruction Format Design

The selection of instructions to be included in the CPU is also done in this stage. Each of the selected operation is assigned with a unique opcode in order for the processor to differentiate between them. Their operands are then fitted accordingly, depending on the instruction. The instructions are divided into 3 categories – R-Type, I-Type, and J-Type, similar to MIPS architecture.

Below are the figures of the operands field design for all three types of instructions. Note that the instruction is designed to be 16 bits instead of 32 bits to reduce complexity. However, the concept of the educational processor remains similar to MIPS architecture.

15 – 11	10 – 8	7 – 5	4 – 2	1 – 0
Opcode (5)	RS (3)	RT (3)	RD (3)	SA (2)

Figure 6: R-Type instruction format

15 – 11	10 – 8	7 – 5	4 – 0
Opcode (5)	RS (3)	RT (3)	Immediate (5)

Figure 7: I-Type instruction format

15 – 11	10 – 0
Opcode (5)	Target Jump Address (11)

Figure 8: J-Type instruction format

As shown in the figures above, the instruction format is very similar to the instruction format of MIPS architecture. The difference is that in MIPS architecture, the instruction format is 32-bit instead of 16-bit. The number of bits in opcode and in the operand fields is reduced. The function field is present in MIPS architecture, but not in this processor because we do not have as much number of instructions in this project. So, the function field is not needed. So, as a summary, the designed bits are:

- Opcode: 5 bits
- Registers (RS, RT, RD): 3 bits
- Shift Amount (SA): 2 bits
- Immediate (Imm): 5 bits
- Jump Address (JumpAddr): 11 bits

3.4 Tools Required

The tools that are required in this project can be divided into two parts, which is software and hardware. For the software, the tools required are:

- CCS C Compiler v4.106

  This is used to program the microcontroller that will be used for control logic in the CPU.
- PICKit v2.61

  This program is used to write the written program in MPLAB IDE v8.70 into the microcontroller.
- ActivePerl v5.12

  ActivePerl is needed to compile the program written in Perl language.
- Microsoft Visual Studio 2010

  This program is needed for creating the user interface that will interact with the educational processor.
- Quartus v10.1

  This software will be used mostly for simulation.
- CadSoft Eagle Professional v5.6.0

  Eagle is used to design PCB boards.

The hardware needed in this project can easily be obtained in the Electrical & Electronics Store at Block 22. All hardware needed is listed below:

- Microprocessor
- TTL gates (includes ALU, registers, RAM, multiplexers, and others)
- PCB boards

3.5 Instructions List

This educational processor is capable of executing 16 types of instructions. The instructions are listed below, together with its corresponding opcode, operands, and how it operates.

Name	Mnemonic	Type	Operation	Opcode
Add	add	R	`add $rd, $rs, $rt` R[rd]=R[rs]+R[rt]	00001
Add Immediate	addi	I	`addi $rt, $rs, Imm` R[rt]=R[rs]+ZeroExtImm	00010
Subtract	sub	R	`sub $rd, $rs, $rt` R[rd]=R[rs]-R[rt]	00011
And	and	R	`and $rd, $rs, $rt` R[rd]=R[rs]&R[rt]	00100
And Immediate	andi	I	`andi $rt, $rs, Imm` R[rt]=R[rs]&ZeroExtImm	00101
Or	or	R	`or $rd, $rs, $rt` R[rd]=R[rs]|R[rt]	00110
Or Immediate	ori	I	`ori $rt, $rs, Imm` R[rt]=R[rs]|ZeroExtImm	00111
Xor	xor	R	`xor $rd, $rs, $rt` R[rd]=R[rs]^R[rt]	01000
Xor Immediate	xori	I	`xori $rt, $rs, Imm` R[rt]=R[rs]^ZeroExtImm	01001
Shift Left Logical	sll	R	`sll $rd, $rs, shamt` R[rd]=R[rs]<<shamt	01010
Shift Right Logical	srl	R	`srl $rd, $rs, shamt` R[rd]=R[rs]>>shamt	01011
Store Word	sw	I	`sw $rt, $rs, Imm` M[R[rs]+ZeroExtImm]=R[rt]	01100
Load Word	lw	I	`lw $rt, $rs, Imm` R[rt]=M[R[rs]+ZeroExtImm]	01101

Branch On Equal	beq	I	`beq $rs, $rt, BranchAddr`
			if(R[rs]==R[rt])
			PC=PC+4+BranchAddr
				01110

Branch On Not Equal	bne	I	`bne $rs, $rt, BranchAddr`
			if(R[rs]!=R[rt])
			PC=PC+4+BranchAddr
				01111

Jump	j	J	`j JumpAddr` PC=JumpAddr	10000

Table 1: Instructions List

3.6 Datapath

The datapath of this EduCPU is designed so that it can perform the instructions listed before. The whole datapath of this educational processor is shown in Appendix B.

3.7 Graphical User Interface

A program is written using Perl and Visual Basic 2010. This software is able to compile assembly language that is inputted by user and compile it into machine language. It is equipped with error-checking codes to prevent errors during transmission. The program is also able to transmit the machine codes into the educational processor hardware and retrieve back the relevant data from the educational processor. Besides transmitting the machine code to the actual hardware, the program is able to simulate written assembly code, and display the flow of data. More details about this program are covered in Chapter 4.

3.8 Project Duration

In order to effectively monitor the progress of this project, a Gantt chart has been constructed. The Gantt chart is included in Appendix A.

CHAPTER 4

RESULTS AND DISCUSSION

4.1 Graphical User Interface

A program designed to run in Windows is programmed using Microsoft Visual Studio 2010 and also Perl programming language. This program is called EduCPU. The Visual Basic source code is attached in Appendix F. It is designed to compile from assembly language input by user to the machine code that follows the instruction format designed. EduCPU's features are:

- Compiles assembly language code into machine language code
- Simulates the written assembly code
- Displays values in registers
- Changes values in registers
- Displays values in memory
- Displays data that flows in the datapath

EduCPU consists of 5 main windows. First is the main view. This is where the user needs to put the assembly code. Figure 9 shows the main view window.

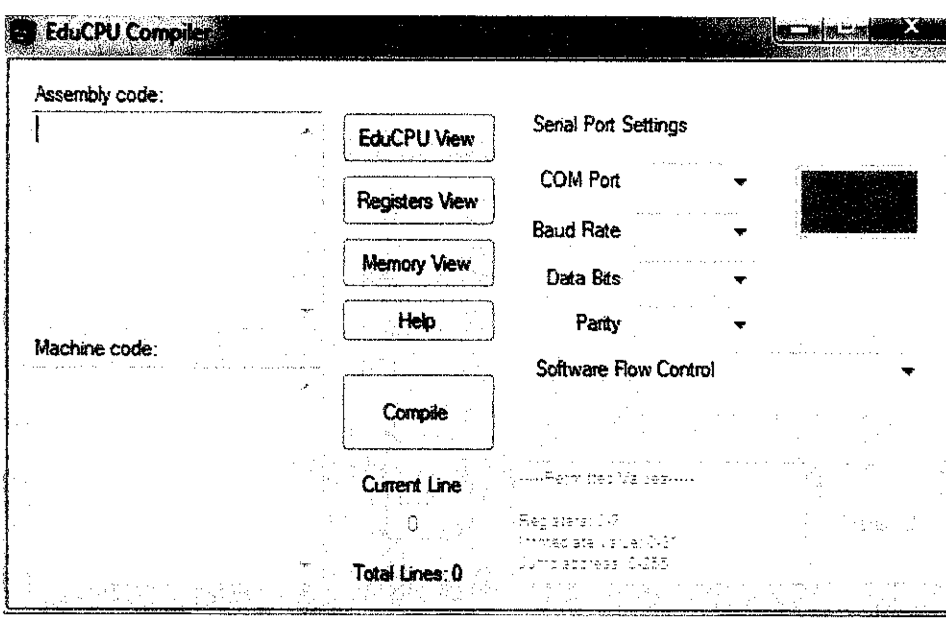


Figure 9: EduCPU Main View

The program is equipped with error-checking codes. This means, if the user enters a wrong command, wrong syntax, or invalid register/immediate values, the compiler will tell the user there has been an error in the code.

When the user puts the assembly code in the designated field and presses 'Compile', the program will check the code for any errors. If there are errors, the text box at the right side will display the error and the 'Transmit' button is disabled. See below for example.

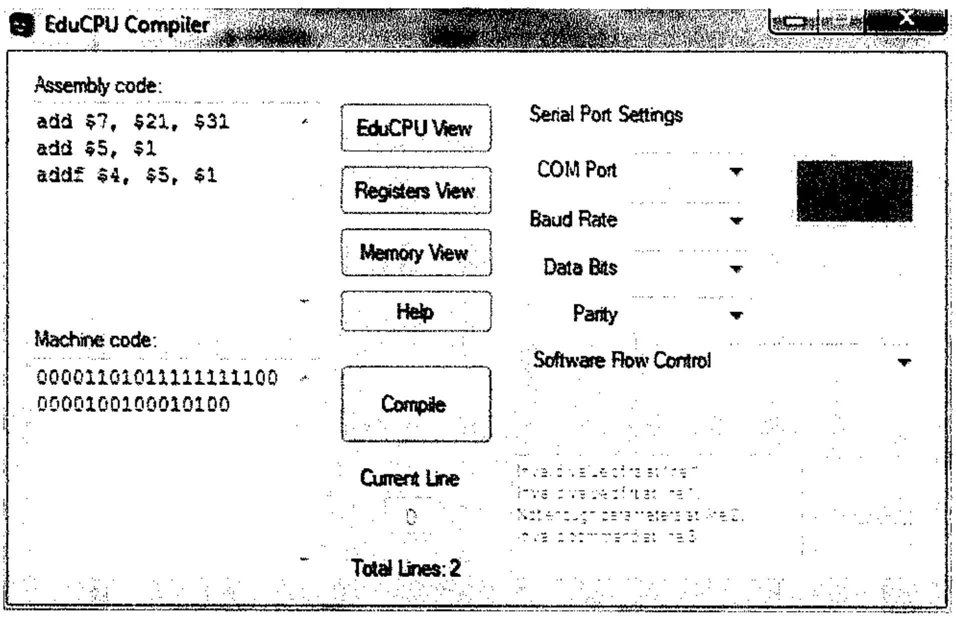


Figure 10: Example of code with errors

In Figure 10, the user puts assembly code with some errors listed below:

- Value of RS and RT at line 1 is too large. We only have 8 registers, so the maximum value should be $7.
- At line 2, the command 'add' should have 3 operands. The user only puts 2 operands, thus generating the error code.
- At line 3, there is no such command 'addf'.

If the user puts code with no syntax error, the bottom box will display "Code OK. Proceed with transmission". The 'Transmit' button will be enabled and the user can now proceed with the simulation as shown in Figure 11.

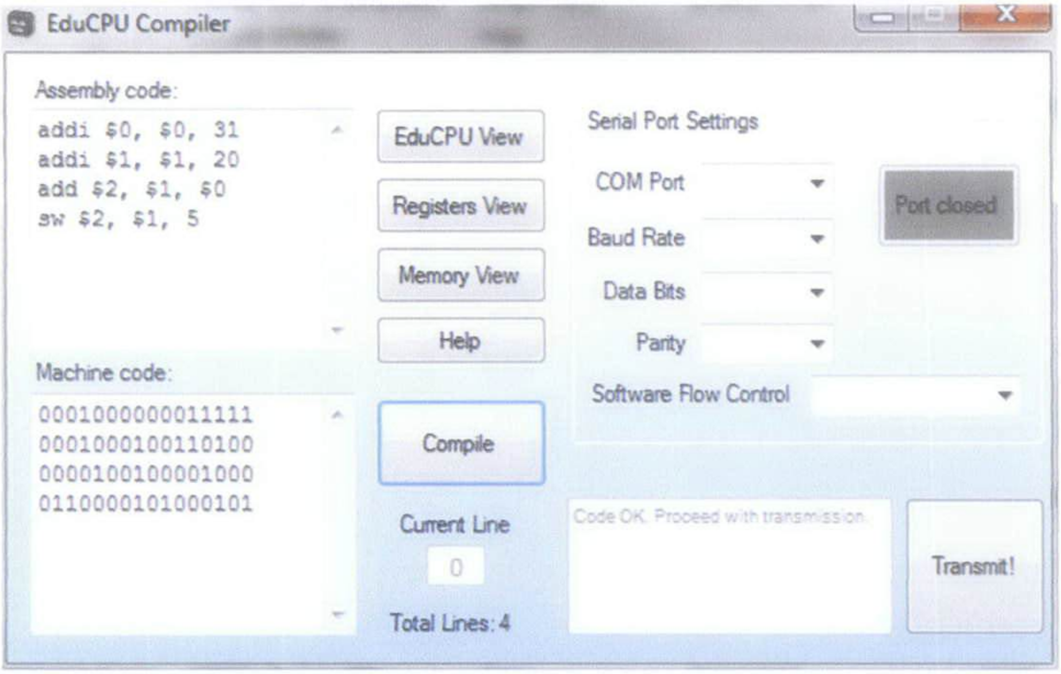


Figure 11: Code without any syntax error

Before proceeding with the simulation by pressing 'Transmit' button, user should first click 'EduCPU View' to view the datapath, 'Registers View' to view the values in registers, and 'Memory View' to view the values in memory. The 3 windows are shown in Figure 12, Figure 13, and Figure 14 respectively.

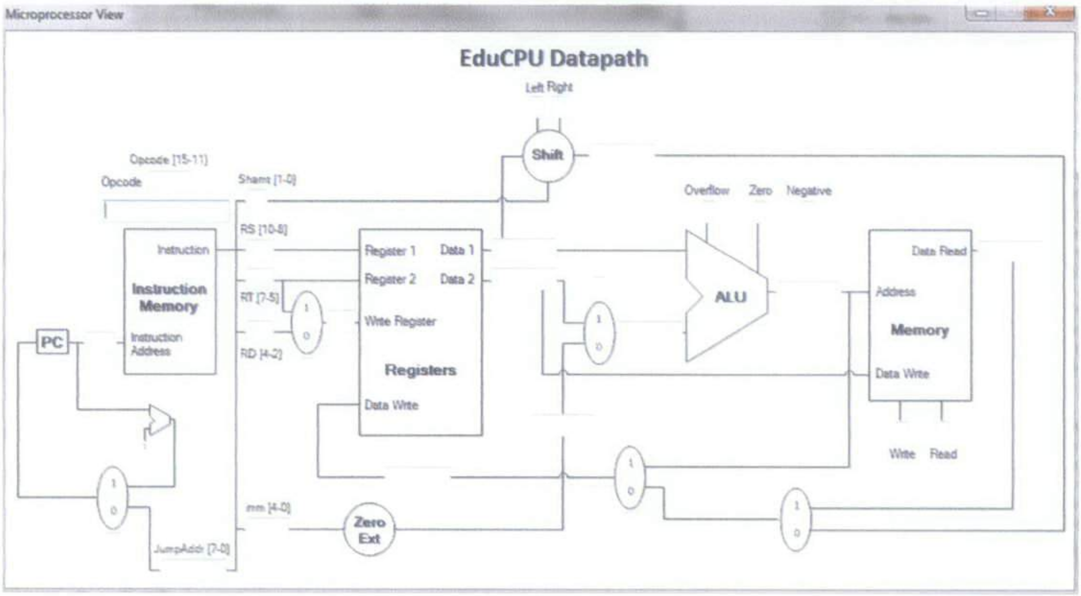


Figure 12: EduCPU Datapath View

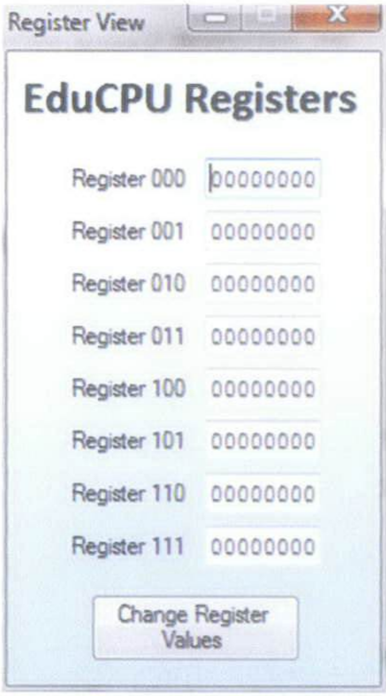


Figure 13: Register View

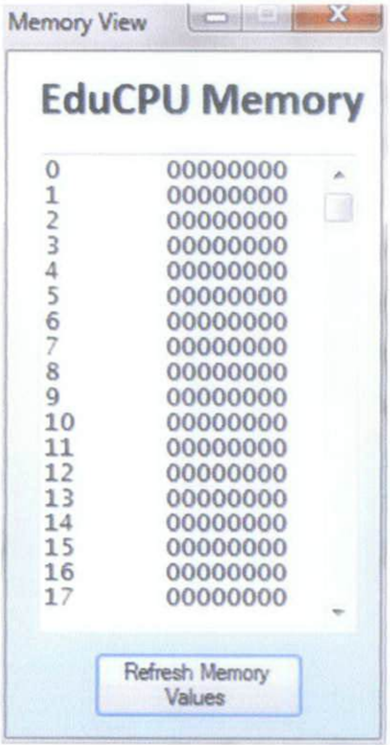
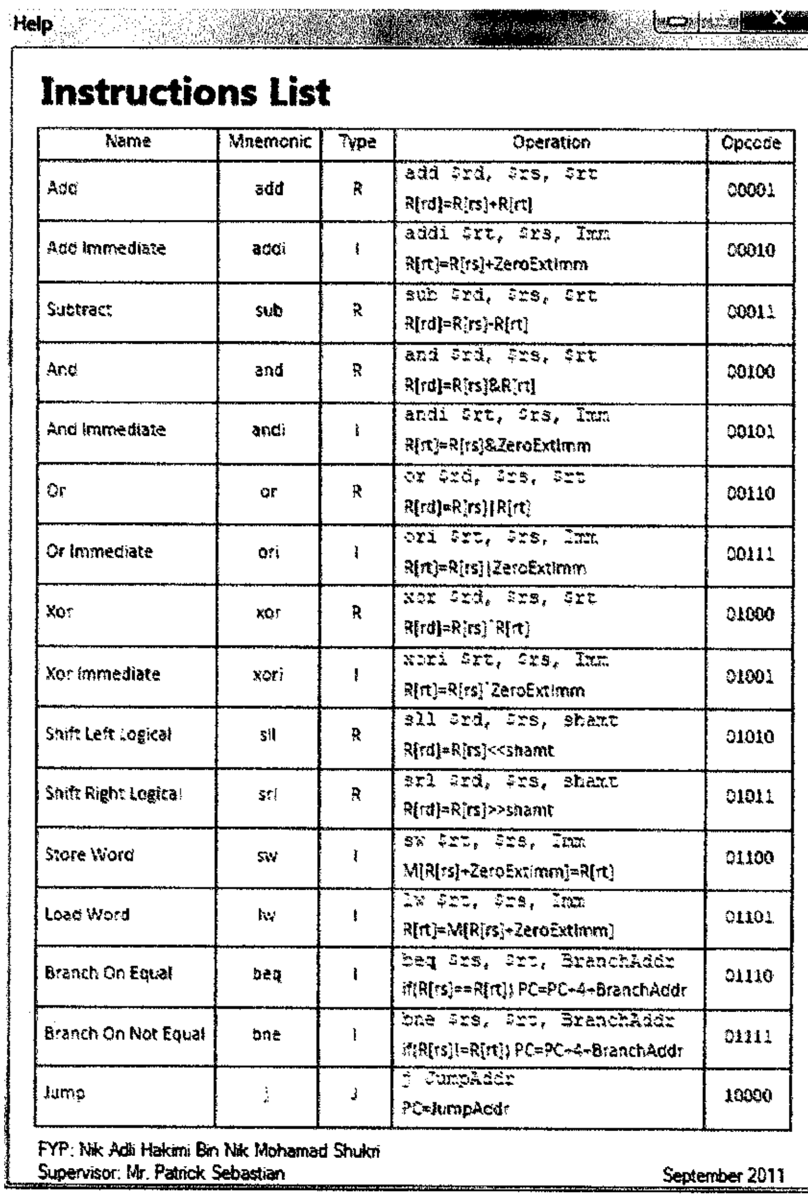


Figure 14: Memory View

We can directly change the values in the registers during simulation using the Register View window. After changing the values, press the 'Change Register Values' button to save the changes.

As guide for users, they can click the 'Help' button to display the list of instructions and the correct syntax on how to use them. Note that the instructions are very similar to MIPS instructions. After clicking the 'Help' button, the window as shown in Figure 15 should pop out.


Help

Instructions List

Name	Mnemonic	Type	Operation	Opcode
Add	add	R	add $rd, $rs, $rt R[rd]=R[rs]+R[rt]	00001
Add Immediate	addi	I	addi $rt, $rs, Imm R[rt]=R[rs]+ZeroExtImm	00010
Subtract	sub	R	sub $rd, $rs, $rt R[rd]=R[rs]-R[rt]	00011
And	and	R	and $rd, $rs, $rt R[rd]=R[rs]&R[rt]	00100
And Immediate	andi	I	andi $rt, $rs, Imm R[rt]=R[rs]&ZeroExtImm	00101
Or	or	R	or $rd, $rs, $rt R[rd]=R[rs]|R[rt]	00110
Or Immediate	ori	I	ori $rt, $rs, Imm R[rt]=R[rs]|ZeroExtImm	00111
Xor	xor	R	xor $rd, $rs, $rt R[rd]=R[rs]^R[rt]	01000
Xor Immediate	xori	I	xori $rt, $rs, Imm R[rt]=R[rs]^ZeroExtImm	01001
Shift Left Logical	sll	R	sll $rd, $rs, shamt R[rd]=R[rs]<<shamt	01010
Shift Right Logical	srl	R	srl $rd, $rs, shamt R[rd]=R[rs]>>shamt	01011
Store Word	sw	I	sw $rt, $rs, Imm M[R[rs]+ZeroExtImm]=R[rt]	01100
Load Word	lw	I	lw $rt, $rs, Imm R[rt]=M[R[rs]+ZeroExtImm]	01101
Branch On Equal	beq	I	beq $rs, $rt, BranchAddr if(R[rs]==R[rt]) PC=PC+4+BranchAddr	01110
Branch On Not Equal	bne	I	bne $rs, $rt, BranchAddr if(R[rs]!=R[rt]) PC=PC+4+BranchAddr	01111
Jump	j	J	j JumpAddr PC=JumpAddr	10000

FYP: Nik Adli Hakimi Bin Nik Mohamad Shukri
Supervisor: Mr. Patrick Sebastian
September 2011

Figure 15: Help Window

4.2 Compiling

As can be seen in Figure 11 in previous pages, the assembly code is automatically compiled into machine code. The exact process can be seen through examples in section 4.3 .

4.2.1 R-Type

```
add $2, $1, $0
```

R-Type instructions' detailed information is shown in Chapter 3.5. For the above instruction, the 'add' instruction is translated into 00001 (the opcode). Then, the register RS which is $1 is translated into its binary equivalent, which is 001. Register RT, $0 is translated into 000. Register RS, $2, is translated to 010. Finally, since no shift operation is involved, the shift amount (2 bits) is set to 00.

Note that all registers are designed to be 3 bits. So, all register values are translated into 3 bits. This process generates the following machine code:

```
0000100100001000
```

4.2.2 I-Type

```
addi $1, $2, 20
```

I-Type instructions require 3 operands. For 'addi' instruction, the opcode is 00010. Then, register RS, $2 is translated into 010. Register RT, $1 is translated into 001. Finally, the immediate value, 20 is translated into 10100. Immediate value is designed to be 5 bits. So, this generates the following machine code:

```
0001001000110100
```

4.2.3 J-Type

```
j 3
```

J-Type instructions require only 1 operand, which is JumpAddr (Jump Address). The opcode for instruction 'j' is 10000. Then, the JumpAddr, which is valued at 3, is translated into 00000000011. This process generates the following machine code:

```
1000000000000011
```

4.3 Simulation: Test Code 1 – Adding values

```
addi $0, $0, 31
addi $1, $1, 20
add $2, $1, $0
sw $2, $1, 5
```

This test code will basically add value 31 to register 0, add value 20 to register 1, and then add the values in register 0 and register 1 into register 2. Value in register 2 will then be stored into memory location 25. We can track which data goes where exactly by observing the Datapath View, Register View, and Memory View. The compiled test code can be seen in Figure 11.

After pressing 'Transmit' button once, we can see the Current Line in Main View changes to 1. The values in Datapath View and Register View are also updated as seen in Figure 16 and Figure 17. We can see how data flows into the CPU and also real time register values.

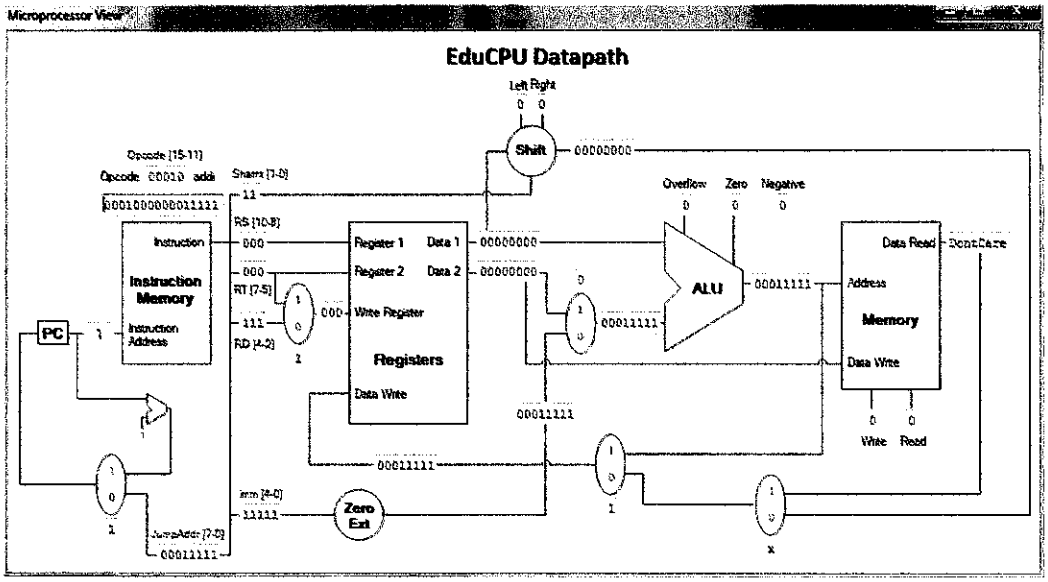


Figure 16: Test Code 1, Line 1 Datapath View

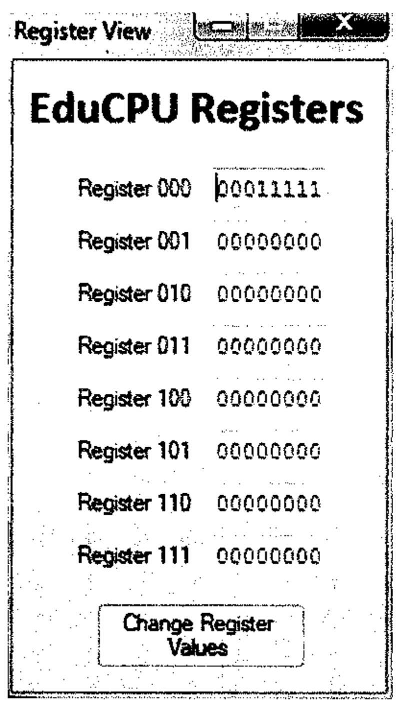


Figure 17: Test Code 1, Line 1 Register View

After line 1 is simulated (we can see current line in Datapath View, at PC (program counter) value), register 0 [000] is loaded with value 31 [00011111]. This happens because value in register 0 is given by summation of current value in register 0 (which is currently 0) and immediate value of 31. By pressing 'Transmit' button again, line 2 is simulated. Line 2 is similar to line 1 – they both use 'addi' command. After pressing 'Transmit' button for the third time, line 3 is simulated, as shown in Figure 18 and Figure 19.

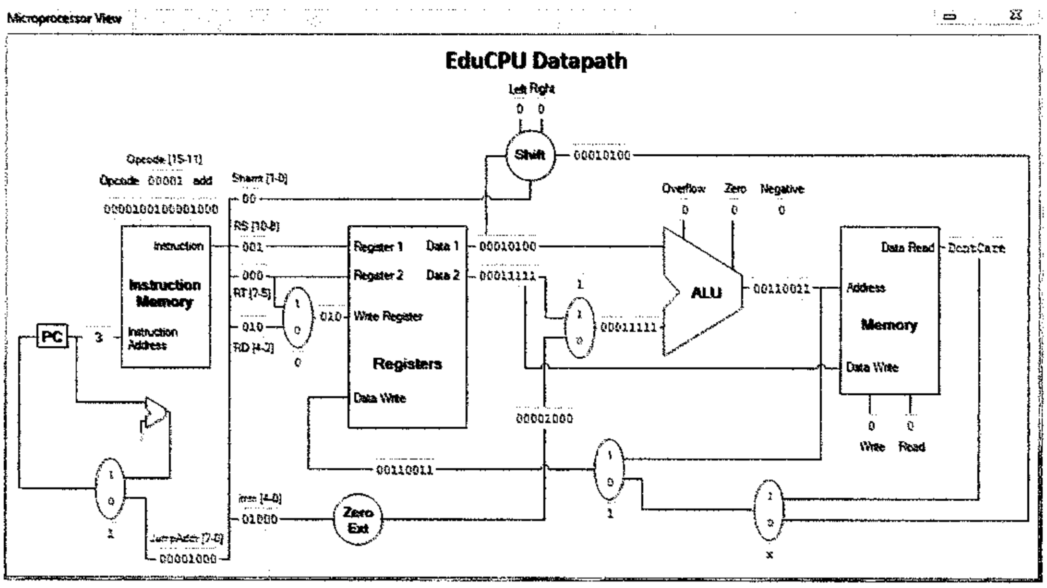


Figure 18: Test Code 1, Line 3 Datapath View

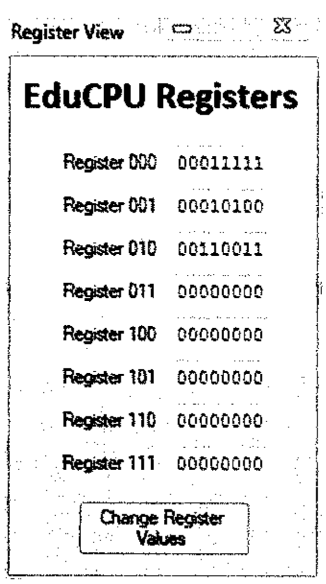


Figure 19: Test Code 1, Line 3 Register View

After simulating line 3, we can see the updated values in Register View. Register 1 [001] is loaded with value 20 [00010100] during simulation of line 2. Register 2 [010] now has value 51 [00110011], which is the summation of values in register 0 and register 1. We can see the in Datapath View around the Registers, Data

1 comes from register [001], and Data 2 comes from register [000]. Both these data flows into the ALU, and the result of [00110011] comes out from the ALU. The data is written back into register [010], which is register 2. Now, we simulate line 4 of the code.

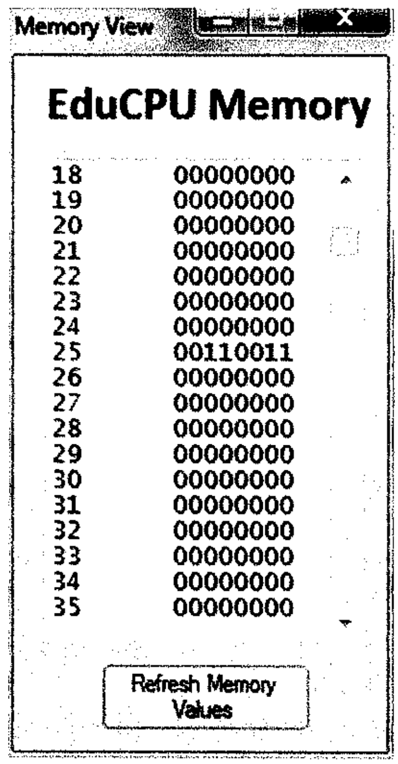


Figure 20: Test Code 1, Line 4 Memory View

According to the command executed, which is [`sw $2, $1, 5`], the value in register 2 [010] will be stored into memory value register 1 [001] + 5. Since the value in register 1 is 20 [00010100], when added with 5, the result is 25. We can see the updated memory values by pressing 'Refresh Memory Values' button. So, the value in register 2, which is 51 [00110011] is stored in memory location 25 as shown in Figure 20 above. Now the code has completed the simulation. The 'Transmit' button in Main View is now automatically disabled again.

4.4 Simulation: Test Code 2 – Multiplying values

```
addi $0, $0, 3
addi $1, $1, 6
addi $2, $2, 1
add $3, $1, $3
sub $0, $0, $2
bne $0, $4, $4
```

In this test code, we are going to do a 6 × 3 multiplication. The answer will be saved into register 3. The method of doing this is we are going to add the value 6 with itself 3 times.

In the first two lines, we are just putting the numbers to be multiplied in the registers. Value 3 [00000011] is saved into register 0 and value 6 [00000110] is saved into register 1. In third line, value 1 [00000001] is saved into register 2. The purpose is so that we can subtract one from the counter in each loop.

In the fourth line, the data in register 1 is added to the data in register 3 and stored into register 3. This action is shown in Figure 21 and Figure 22.

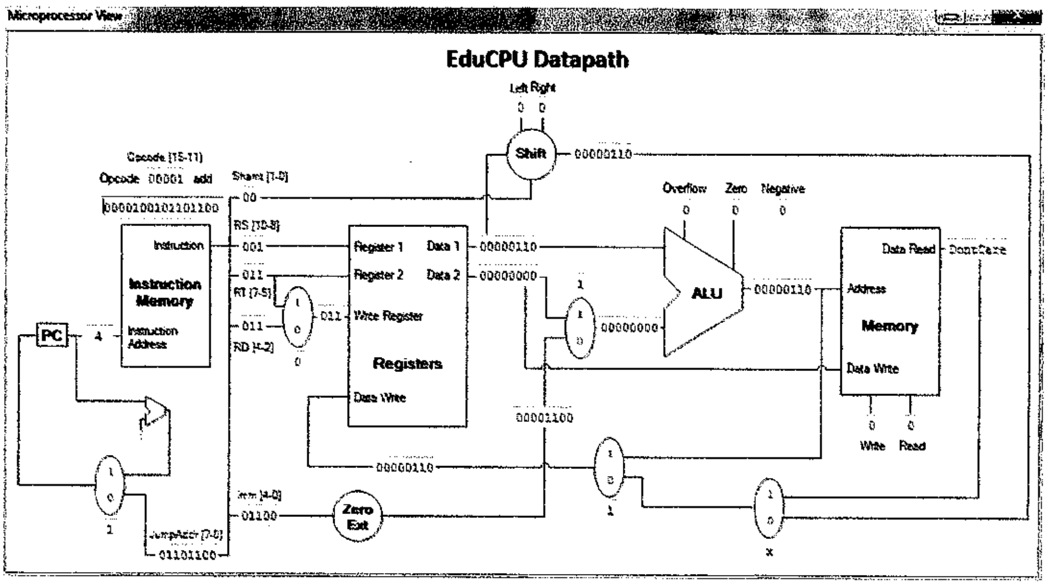


Figure 21: Reg 3 = Reg 1 + Reg 3

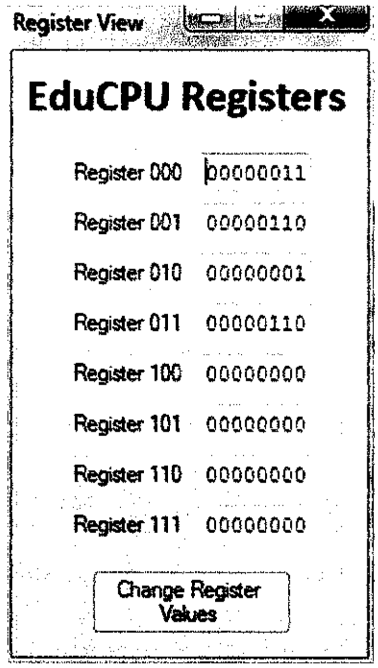


Figure 22: Registers view after line 4 is transmitted

After the value 6 is added once, the value in register 0 is subtracted by one. The value in register 0 represents the loop counter. Once it reaches 0, the program will no longer add the value 6 into register 3. Figure 23 shows the registers view after line 5 has been transmitted:

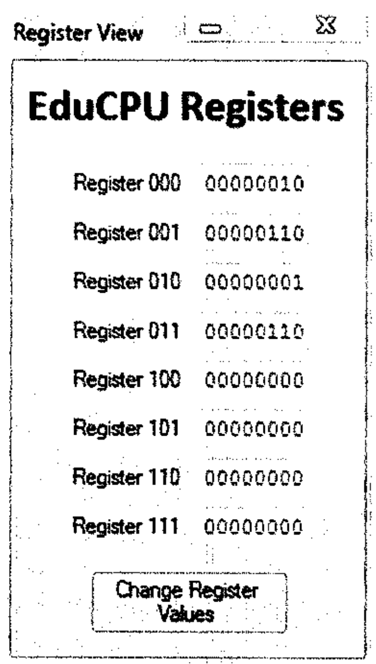


Figure 23: Registers view after line 5 is transmitted

As we can see in Figure 23, the value in register 0 is decreased by one. The last line checks if the value in register 0 is equal to value in register 4 or not. If it is not equal, the program will go back to line 4 of the code. Now, the value in register 0 is 2 [00000010] and the value in register 4 is 0 [00000000]. It is not equal (Zero flag not raised to 1), means the program will go back to line 4. Figure 24 shows the datapath view after transmitting the last line. Note that the zero flag is zero:

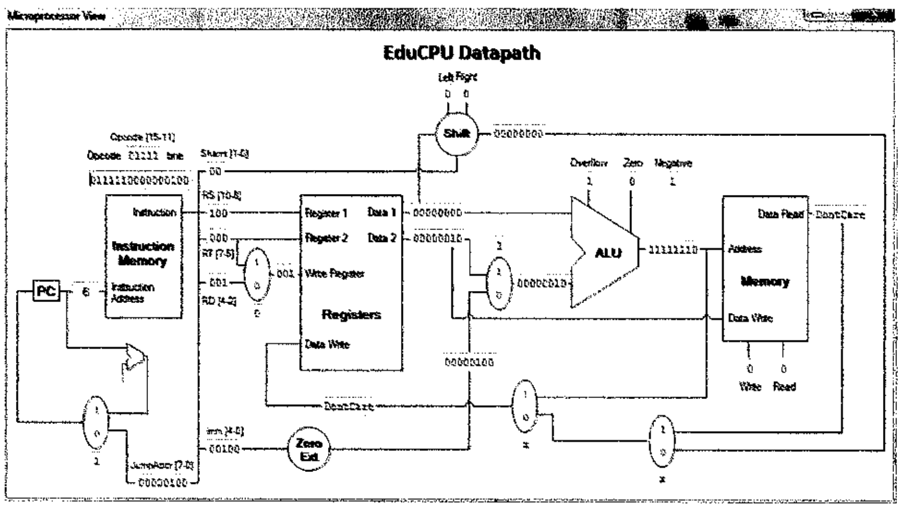


Figure 24: Datapath view after transmitting line 6

The program now goes back to line 4. The main view, datapath view, and registers view is shown in Figure 25, Figure 26, and Figure 27 respectively.

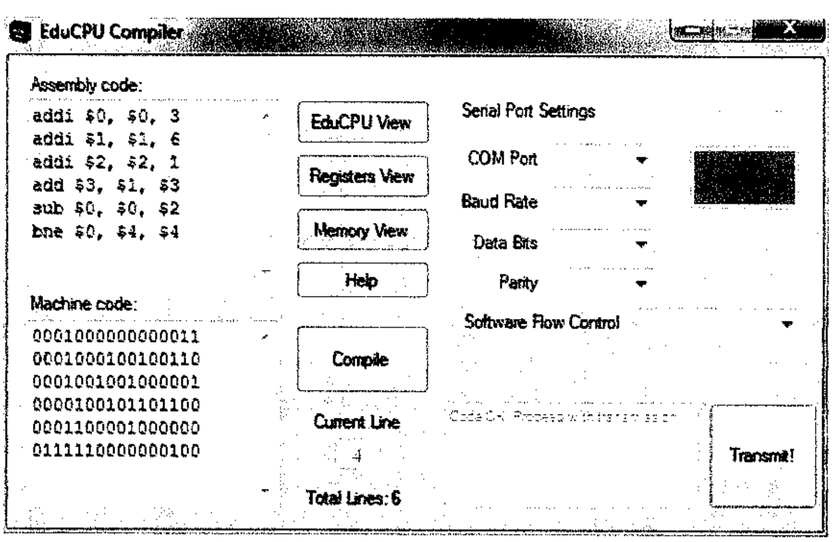


Figure 25: Main view after transmitting line 4 for second time

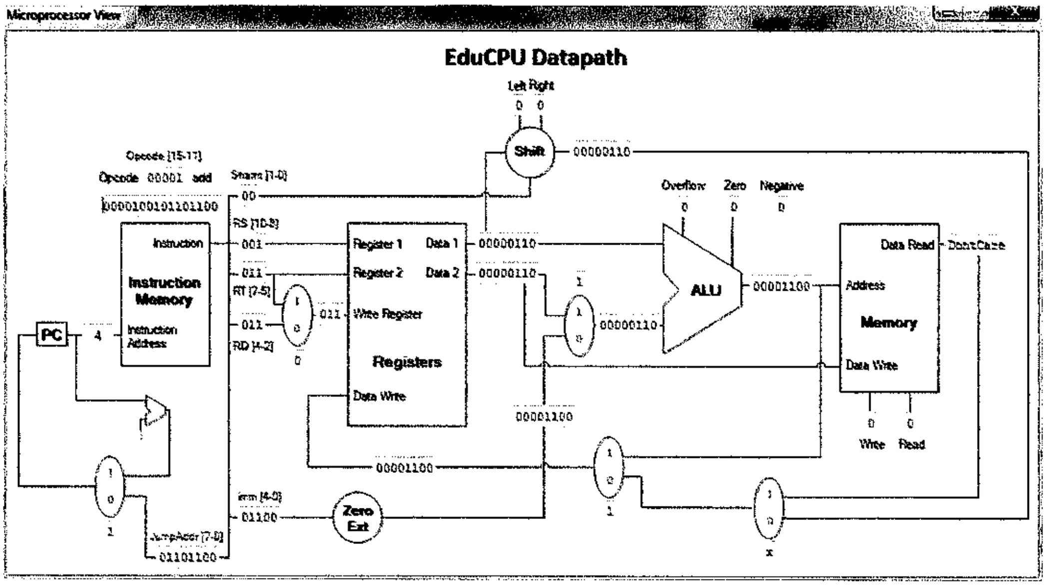


Figure 26: Datapath view after transmitting line 4 for second time

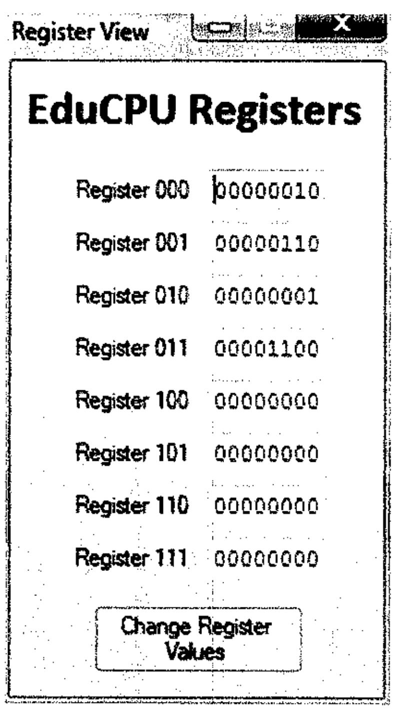


Figure 27: Registers view after transmitting line 4 for second time

We can see the value in register 3 is now 12 [00001100]. Value in register 0 will be decreased by one again, and then 'bne' instruction will check if value in register 0 is equal to value in register 4. It is still not equal because at this point of time, the value in register 0 is 1 [00000001] and value in register 4 is 0 [00000000]. The program will go back to line 4 again for transmission.

In line 4, the value 6 will be added again to register 3, making the value in register 6 to be 18 [00010010]. In line 5, the register 0 will contain the value 0 [00000000] because the previous data, which was 1, is decreased again by one. In line 6, 'bne' instruction will check again if value in register 0 is equal to the value in register 4. If they are equal, the zero flag will be raised. Now we can see that they are equal, denoted by the zero flag raised to 1. This is shown in Figure 28.

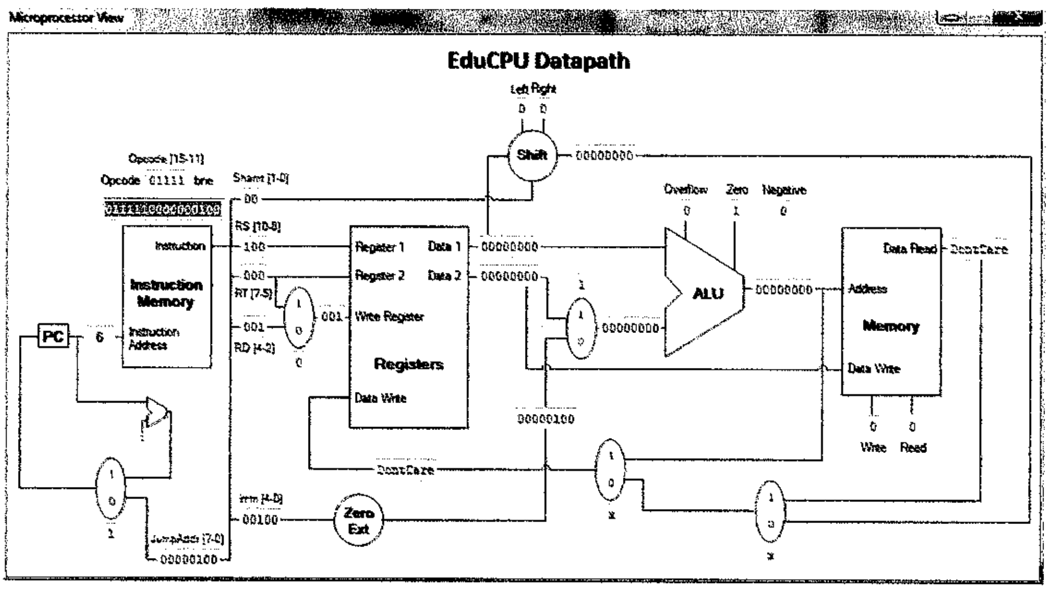


Figure 28: Datapath view after transmitting the last line of the code

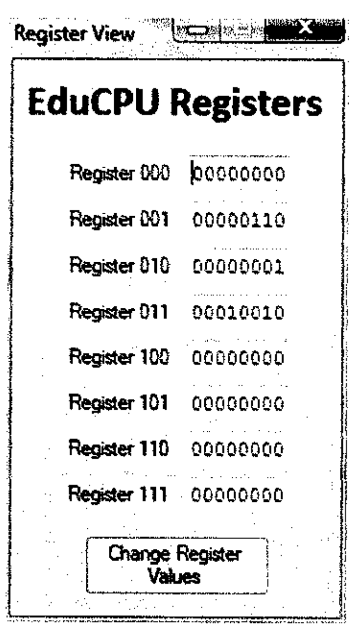


Figure 29: Registers view after transmitting the last line of the code

Note that the program is complete at this point. We have the value in register 0 to be 0 [00000000], register 1 to be 6 [00000110], register 2 to be 1 [00000001] and register 3 to be 18 [00010010] (our wanted final answer). The register values are shown in Figure 29. The 'Transmit' button is also disabled now, indicating that the program has reached the end, as shown in Figure 30.

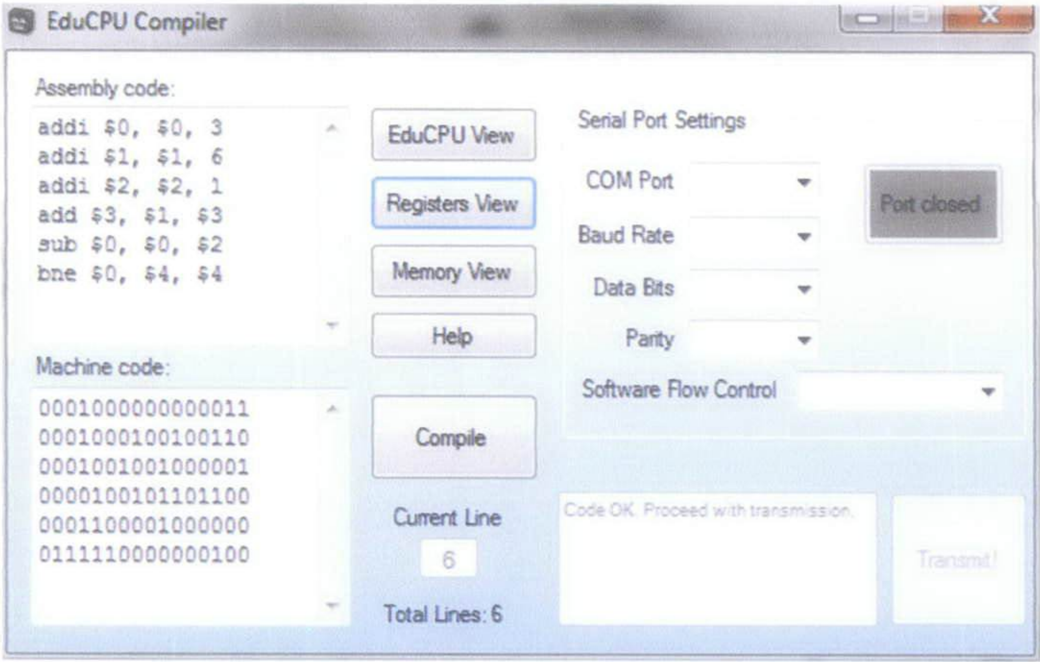


Figure 30: Main view after transmitting the last line of the code

4.5 Transmitting machine code through serial communication

The machine code compiled by the graphical user interface can be transmitted to the actual hardware through serial communication as shown in Figure 31 below.

Figure 31: Connection between PC and hardware

Since there will be a delay during transmission, the user has to wait approximately 2 seconds before the data transmission can complete successfully. To make sure that the simulation software is synchronized with the hardware result, the simulation result is also delayed by approximately 2 seconds if a port is opened. If no port is opened, the simulation will show results instantaneously.

In order to transmit using correct port, the hardware needs to be connected to a serial port of the computer using a serial cable. Then, the correct port must be selected using the graphical user interface as shown in Figure 32 below.

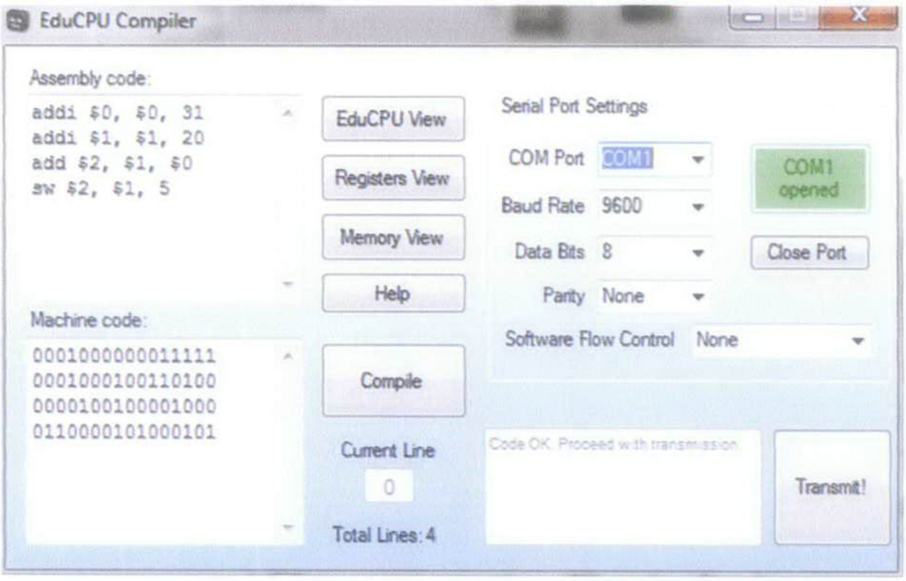


Figure 32: Choosing the correct serial port

4.6 Hardware

The circuit design has been finished according to the datapath design shown in Appendix B. The circuit design is done using CadSoft Eagle Professional v5.6.0. It is shown in Appendix C. However, the actual implementation is that the circuit is divided into 5 smaller parts for easier fabrication. The divided PCB parts are shown in Appendix D.

Data is stored in register files. 4 register files are used in the designed circuit. This is because the educational processor is designed with 8 8-bit registers. Each register file only stores 4 4-bit register, so we need 4 register files in order to fulfill the requirement of the design. Arithmetic logic unit (ALU) is used to compute certain instructions such as add, sub, or, xor, and many more. 2 ALUs are used because each ALU can only process 4 bits of data. Since the whole processor is designed for 8-bit data, we need 2 ALUs to complete the instructions given.

Depending on the instruction, ALU, registers, and memory will need to receive data from different parts of the circuit. The multiplexers are used to control this data flow. 3-state buffers are used to disconnect certain data whenever it is not needed, so that the data will not interfere with data coming from another part of the circuit. Memory is used to store data temporarily. The memory is able to store a total of 2048 8-bit data.

Two microcontrollers will be used to send the control logics to all the other parts of the circuit. They will receive instructions from a connected computer through serial communication. The software will send the instruction bits to the microcontroller serially, and then the microcontrollers will send the control logics to each component on the circuit in order to execute the instructions properly. The data flowing in the circuit will be shown by LEDs on the circuit for easy reference.

The actual Educational Processor hardware with 16-bit instruction and 8-bit data has been constructed and is shown in Figure 33.

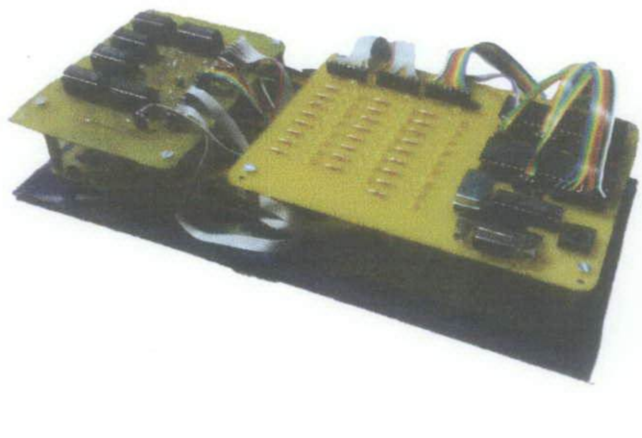


Figure 33: Hardware of Educational Processor

The hardware will perform the same functions as the simulation software, with some differences. The differences are:

- The hardware is unable to process 'srl' and 'sll' instructions due to the absence of parallel shift register.
- Users will not be able to directly change register values using the EduCPU software. Values need to be set using 'addi' instruction.
- The data displayed on the hardware will not be as detailed as in the simulation software. The data displayed will only be at:
    - ALU output / Memory address input
    - Memory output / Memory input
    - Data written back into register

Despite the differences, users will still be able to see what data flows inside the hardware by using multi meter. The LEDs are placed at only significant parts of the circuit for easier implementation.

Upon receiving serial data from the computer, the microcontrollers will interpret the received data and send corresponding control logic signals to the ICs in order to execute the instruction given properly. 2 microcontrollers are used to send logic signals, since the pins needed to control everything could not be covered by only one microcontroller.

If the user does not use any 'srl' or 'sll' instructions or change the values in the register directly, the data displayed at the hardware should be exactly the same with the data shown in the simulation software.

4.7 Discussion

The EduCPU software should be significantly helpful for students who are taking Computer System Architecture course in UTP. The hardware will function just the same as the simulation software, so it is very useful for students who prefer hands-on learning compared to just using the software.

The hardware can be improved some more by adding parallel shift register IC. Due to several constraints such as IC supplies and time constraint, the parallel shift register could not be added to this processor. Other than that, the hardware is working perfectly as intended.

CHAPTER 5

CONCLUSION & RECOMMENDATIONS

5.1 Conclusion

The objective of this project is to develop a simple processor using TTL logic gates for educational purpose. This educational processor will be assisted by software that will communicate with the educational processor through serial communication.

The software can also work on its own without the hardware. It can provide simulation of the codes inputted by the user, and show exactly how MIPS-based processors work. This will definitely help students in learning how processors work, especially in Computer System Architecture course.

5.2 Recommendations

For future work, there are definitely a lot of improvements that can be done to improve the educational processor. Such improvements include:

- The number of instructions can be increased to more than 16 instructions, allowing for more complex programs to be run on the processor.
- Full working CPU capable of running a simple operating system can be constructed. If this is completed, this project can educate students in areas more than just Computer System Architecture, but also in the Operating System, Assembler & Compiler Design and more areas.

REFERENCES

[1] Albert P. Malvino & Jerald A Brown, 3rd edition, "Digital Computer Electronics", "SAP Processor".

[2] Weik, Martin H. (1961), "A Third Survey of Domestic Electronic Digital Computing Systems".

[3] Alan Clements (2006), 4th edition, "Principle of Computer Hardware".

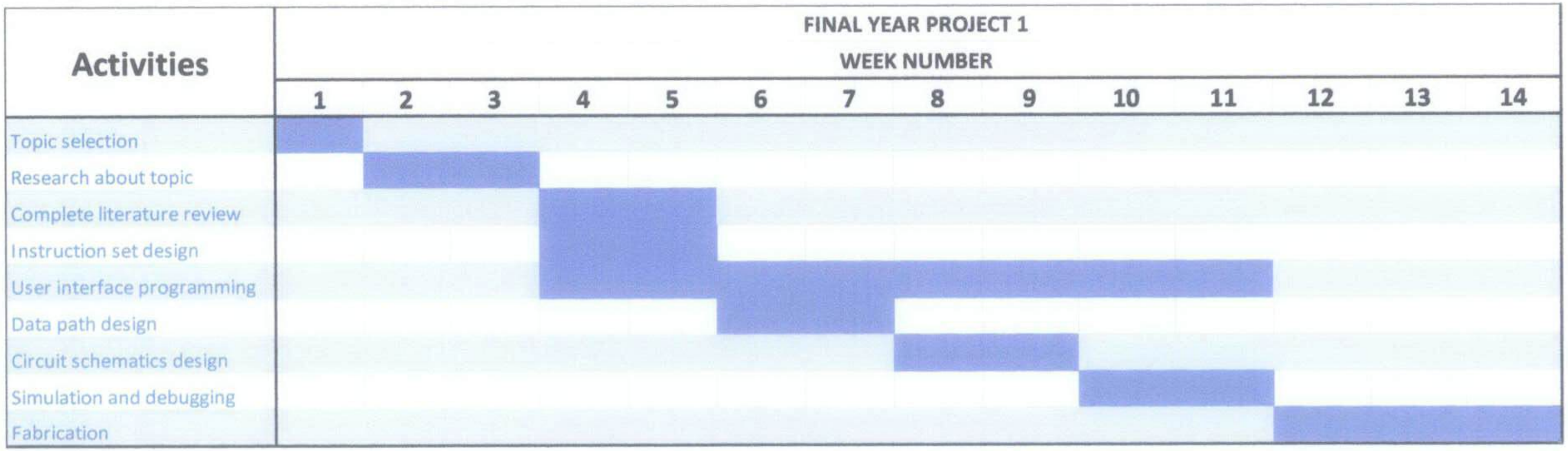
[4] Gary Shute (2007), "MIPS Instruction Coding" http://www.d.umn.edu/~gshute/spimsal/talref.html.

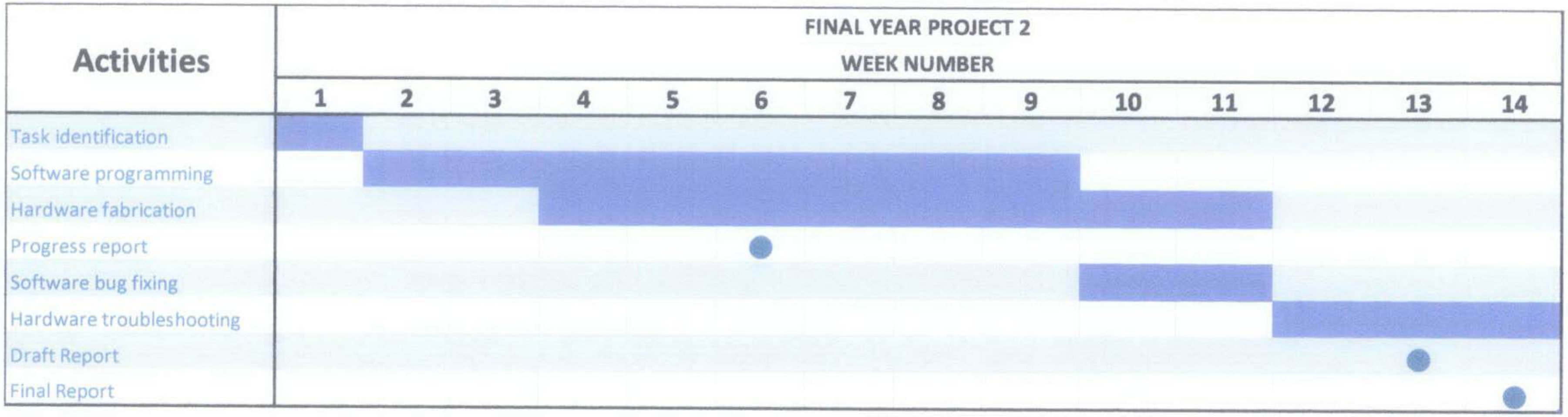
[5] J. Hennessy, N. Jouppi, S. Przybylski, C. Rowen, T. Gross, F. Baskett, and J. Gill (1982), "MIPS: A Microprocessor Architecture".

[6] Hwang, Enoch (2006), "Digital Logic and Microprocessor Design with VHDL". Thomson.

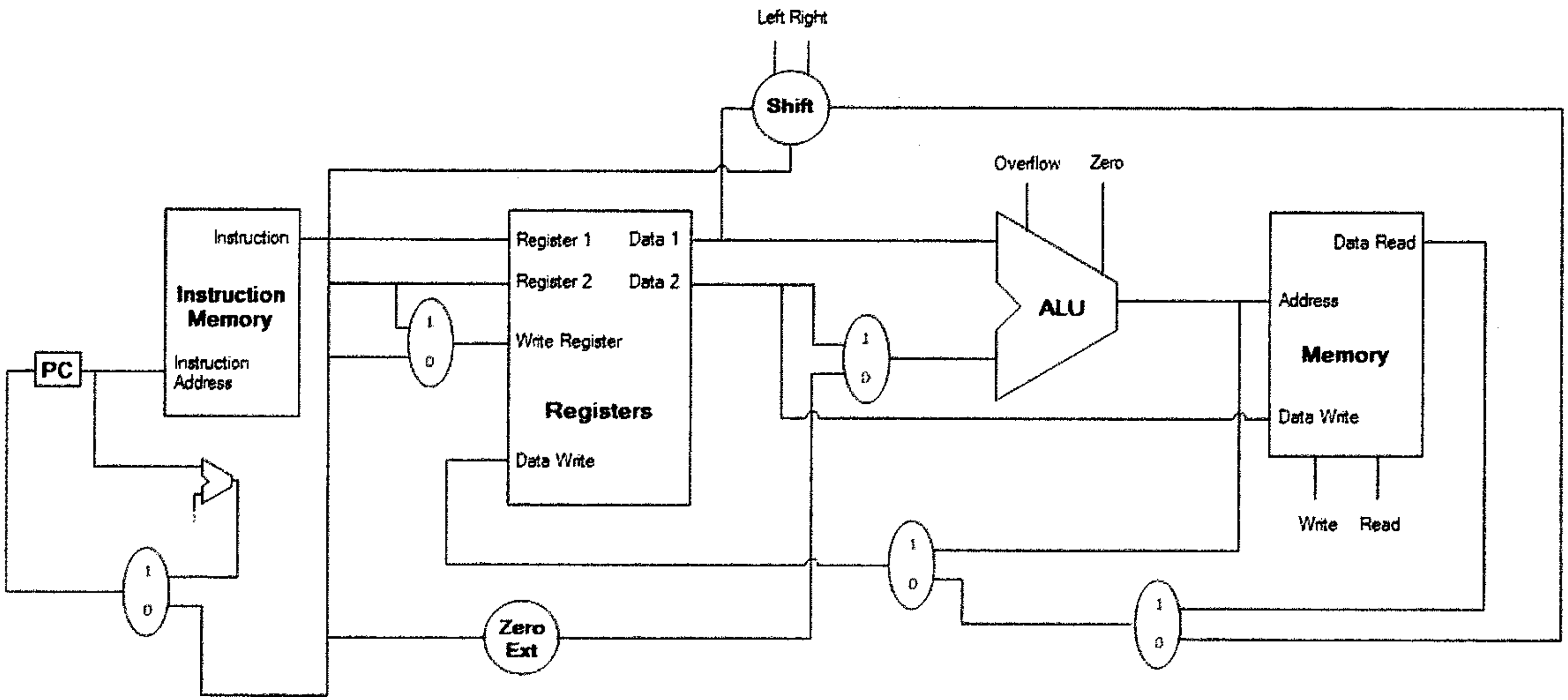
[7] D 'Arcy Becker, Meg Dwyer (1998), "The Impact of Student Verbal/Visual Learning Style Preference on Implementing Groupware in the Classroom".

APPENDIX A – PROJECT GANTT CHART

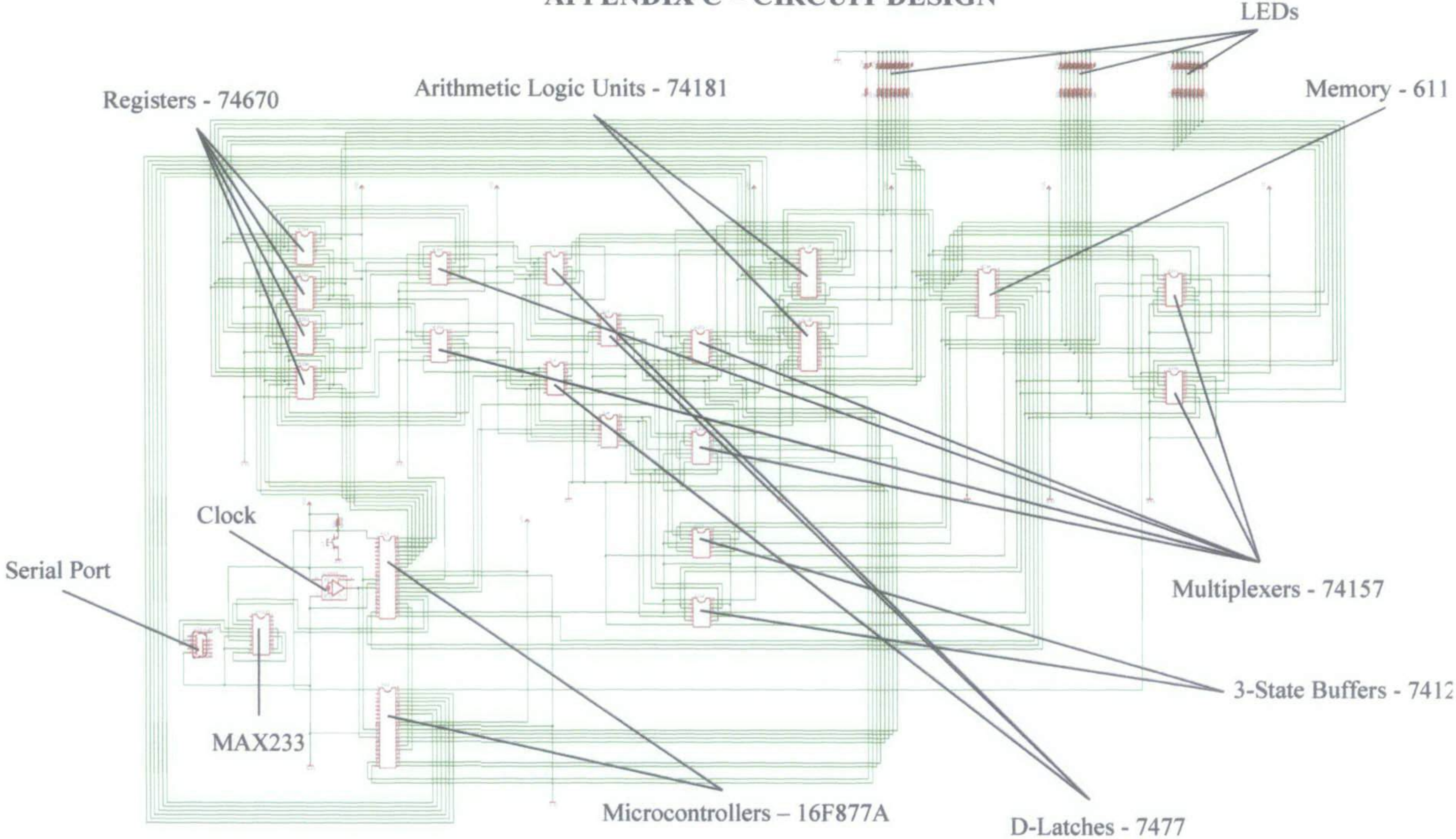



Activities
FINAL YEAR PROJECT 2
WEEK NUMBER
1
2
3
4
5
6
7
8
9
10
11
12
13
14
Task identification
Software programming
Hardware fabrication
Progress report
Software bug fixing
Hardware troubleshooting
Draft Report
Final Report

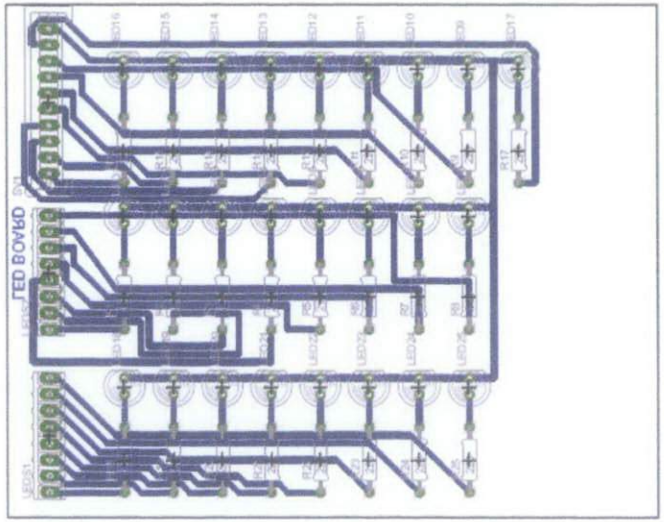
APPENDIX B – DATAPATH DESIGN

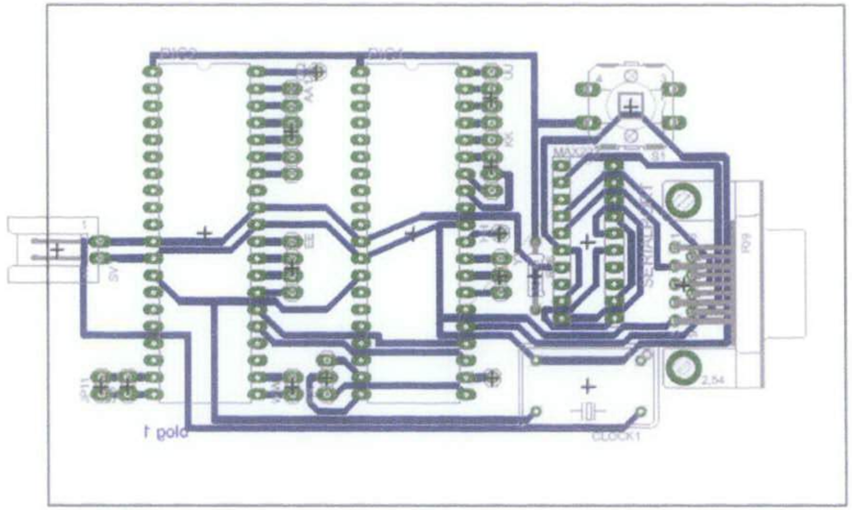


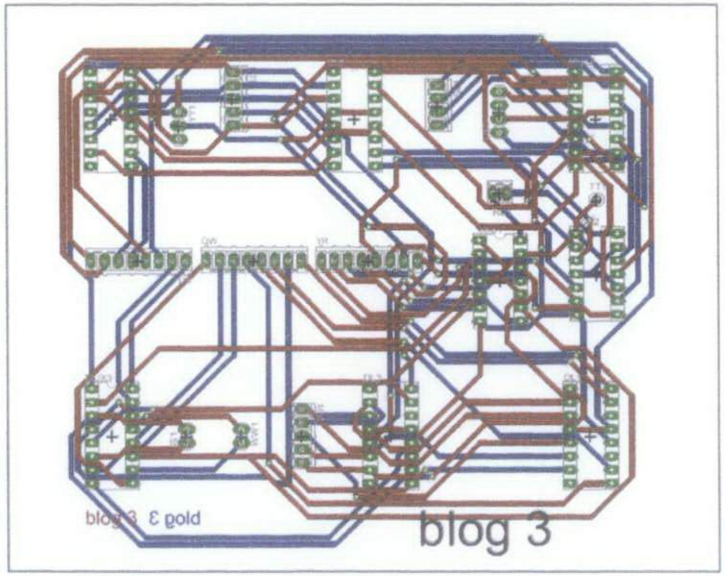
APPENDIX C – CIRCUIT DESIGN

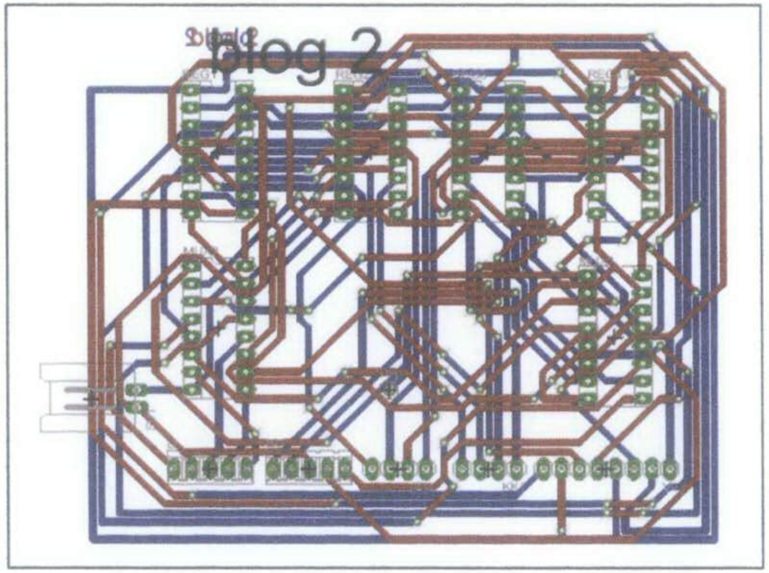


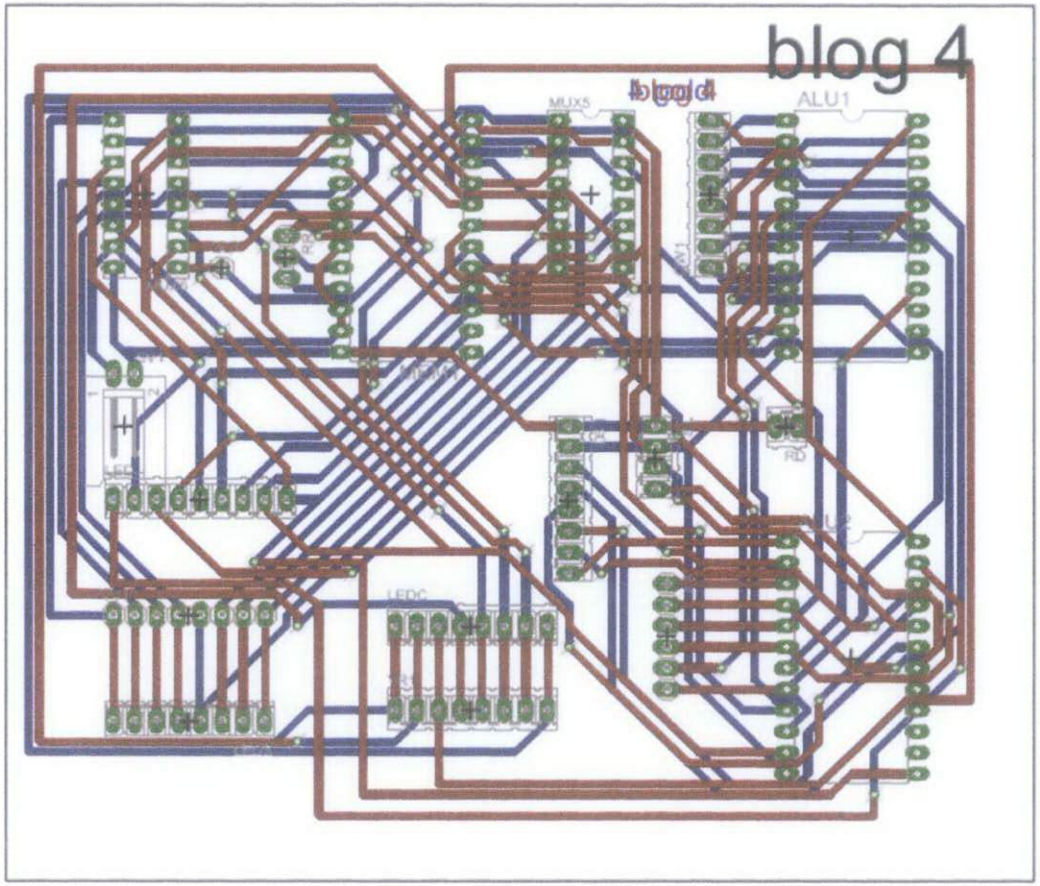
APPENDIX D – SPLIT PCB CIRCUIT DESIGN






blog 3
blog 3


blog 2


blog 4
MUX5
ALU1
LEDC
RD

APPENDIX E – PERL SOURCE CODE

Perl source code for the Compiler Windows application:

```
# Handle input file from every filename in temp file
open(INFILE, "Assembly.txt") || die "Can't find 'Assembly.txt' in current folder. Stopped execution";
@result = <INFILE>;

open(OUTFILE, ">Output.txt") || die "Can't create output file in current folder. File 'Output.txt' may be currently opened. Close
the file and try again. Stopped execution";
@outputfile = <OUTFILE>;

open(ERRFILE, ">Error.txt") || die "Can't create output file in current folder. File 'Error.txt' may be currently opened. Close the
file and try again. Stopped execution";
@errorfile = <ERRFILE>;

if (not defined(@result)) { print ERRFILE "No code written."; goto END;}
	else { }

	$error=0;

for($i=0;$i<=$#result; $i++)
{
	$linenumber=$i+1;   # Line number for error checking
	$type=0;   # Initialize variable type

	@result[$i] =~ s/,/ /g;			# Replace , with blank space
	@result[$i] =~ s/\$/ /g;			# Replace $ with blank space

	@command=split(/\s+/,@result[$i]);		# Split everything
	@command[0] =~ tr/A-Z/a-z/;				# Convert all to lowercase


	if (not defined(@command[0])) { goto END;}		# If blank line, go to end of file (ignore the line)
	else { }

	# Determining the opcode and instruction type, then print the opcode for the given command
	if		(@command[0]=~/^add$/)		{  print  OUTFILE  "00001";  $type='R';
$param=4;}
	elsif	(@command[0]=~/^addi$/)		{ print OUTFILE "00010"; $type='I'; $param=4;}
	elsif	(@command[0]=~/^sub$/)		{ print OUTFILE "00011"; $type='R'; $param=4;}
	elsif	(@command[0]=~/^and$/)		{ print OUTFILE "00100"; $type='R'; $param=4;}
	elsif	(@command[0]=~/^andi$/)		{ print OUTFILE "00101"; $type='I'; $param=4;}
	elsif	(@command[0]=~/^or$/)		{ print OUTFILE "00110"; $type='R'; $param=4;}
	elsif	(@command[0]=~/^ori$/)		{ print OUTFILE "00111"; $type='I'; $param=4;}
	elsif	(@command[0]=~/^xor$/)		{ print OUTFILE "01000"; $type='R'; $param=4;}
	elsif	(@command[0]=~/^xori$/)		{ print OUTFILE "01001"; $type='I'; $param=4;}
	elsif	(@command[0]=~/^sll$/)		{ print OUTFILE "01010"; $type='S'; $param=4;}
	elsif	(@command[0]=~/^srl$/)		{ print OUTFILE "01011"; $type='S'; $param=4;}
	elsif	(@command[0]=~/^sw$/)		{ print OUTFILE "01100"; $type='I'; $param=4;}
	elsif	(@command[0]=~/^lw$/)		{ print OUTFILE "01101"; $type='I'; $param=4;}
	elsif	(@command[0]=~/^beq$/)		{ print OUTFILE "01110"; $type='I'; $param=4;}
	elsif	(@command[0]=~/^bne$/)		{ print OUTFILE "01111"; $type='I'; $param=4;}
	elsif	(@command[0]=~/^j$/)		{ print OUTFILE "10000"; $type='J'; $param=2;}
	else								{ print ERRFILE "Invalid
command at line $linenumber.\n"; $error=1;}

	if  (not  defined(@command[$param-1]))  {  print  ERRFILE  "Not  enough  parameters  at  line  $linenumber.\n";
$error=1;} # Check if enough parameters
	else { }

	if ($type=~R){
	&register_rs;		# Printing 1st register value (rs)
	&register_rt;		# Printing 2nd register value (rt)
	&register_rd;		# Printing 3rd register value (rd)
	print OUTFILE "00\n";				# Printing 0 shift amount
	}

	elsif ($type=~I){
	&register_rs;		# Printing 1st register value (rs)
	&register_rt_i;		# Printing 2nd register value (rd)
	&immediate;				# Printing immediate value
```

```
		}

		elsif ($type=~S){
		&register_rs;		# Printing 1st register value (rs)
		&register_rt;		# Printing 2nd register value (rt)
		&register_rd;		# Printing 3rd register value (rd)
		&shamt;				# Printing shift amount
		}

		elsif ($type=~J){
		&jumpaddr;					# Printing jump address
		}

		else {}
		}

		if ($error == 0){
			print ERRFILE "Code OK. Proceed with transmission.";
		}
		else { }



sub dec2bin
{
my $str = unpack("B32", pack("N", shift));
$str =~ s/^0+(?=\d)//;
return $str;
}

sub register_rs
{
		if			(@command[2]>7)	{ print ERRFILE "Invalid value of rs at line $linenumber.\n";
$error=1;}
		elsif	(@command[2]<2)	{ print OUTFILE "00";}
		elsif	(@command[2]<4)	{ print OUTFILE "0";}
		else {}
		$regrs = dec2bin(@command[2]);
		print OUTFILE $regrs;
}

sub register_rt
{
		if			(@command[3]>7)	{ print ERRFILE "Invalid value of rt at line $linenumber.\n";
$error=1;}
		elsif	(@command[3]<2)	{ print OUTFILE "00";}
		elsif	(@command[3]<4)	{ print OUTFILE "0";}
		else {}
		$regrt = dec2bin(@command[3]);
		print OUTFILE $regrt;
}

sub register_rt_i
{
		if			(@command[1]>7)	{ print ERRFILE "Invalid value of rt at line $linenumber.\n";
$error=1;}
		elsif	(@command[1]<2)	{ print OUTFILE "00";}
		elsif	(@command[1]<4)	{ print OUTFILE "0";}
		else {}
		$regrt = dec2bin(@command[1]);
		print OUTFILE $regrt;
}

sub register_rd
{
		if			(@command[1]>7)	{ print ERRFILE "Invalid value of rd at line $linenumber.\n";
$error=1;}
		elsif	(@command[1]<2)	{ print OUTFILE "00";}
		elsif	(@command[1]<4)	{ print OUTFILE "0";}
		else {}
		$regrd = dec2bin(@command[1]);
		print OUTFILE $regrd;
}
```

```
sub shamt
{
        if                  (@command[3]>3)  {  print  ERRFILE  "Invalid  value  of  shift  amount  at  line
$linenumber.\n"; $error=1;}
        elsif      (@command[3]<2)   { print OUTFILE "0";}
        else {}
        $shiftvalue = dec2bin(@command[3]);
        print OUTFILE $shiftvalue;
        print OUTFILE "\n";
}

sub immediate
{
        if                  (@command[3]>31)  { print ERRFILE "Invalid value of immediate at line $linenumber.\n";
$error=1;}
        elsif      (@command[3]<2)              { print OUTFILE "0000";}
        elsif      (@command[3]<4)              { print OUTFILE "000";}
        elsif      (@command[3]<8)              { print OUTFILE "00";}
        elsif      (@command[3]<16)  { print OUTFILE "0";}
        else {}
        $imm = dec2bin(@command[3]);
        print OUTFILE $imm;
        print OUTFILE "\n";
}

sub jumpaddr
{
        print OUTFILE "000"; #edit this to be 11 bits later
        if                  (@command[1]>255) {  print  ERRFILE  "Invalid  value  of  jump  address  at  line
$linenumber.\n"; $error=1;}
        elsif      (@command[1]<2)              { print OUTFILE "0000000";}
        elsif      (@command[1]<4)              { print OUTFILE "000000";}
        elsif      (@command[1]<8)              { print OUTFILE "00000";}
        elsif      (@command[1]<16)  { print OUTFILE "0000";}
        elsif      (@command[1]<32)  { print OUTFILE "000";}
        elsif      (@command[1]<64)  { print OUTFILE "00";}
        elsif      (@command[1]<128) { print OUTFILE "0";}
        else {}
        $imm = dec2bin(@command[1]);
        print OUTFILE $imm;
        print OUTFILE "\n";
}

END:
```

APPENDIX F – VISUAL BASIC 2010 SOURCE CODE

This is the code for the software with graphical user interface (EduCPU).

```
Imports System
Imports System.IO.Ports
Imports System.Threading
Imports System.Threading.Thread

Public Class FormMainView

#Region "Initialization"

    Dim WithEvents COMPort As New SerialPort
    Dim TransmitCounter As Integer = 0
    Public Register() As String = {"00000000", "00000000", "00000000",
"00000000", "00000000", "00000000", "00000000", "00000000"}
    Dim OverflowFlag As Integer = 0
    Dim ZeroFlag As Integer = 0
    Dim NegativeFlag As Integer = 0
    Dim ShiftLeft As Integer = 0
    Dim ShiftRight As Integer = 0
    Public Memory As New List(Of String)

#End Region

#Region "Submit Button"

    Private Sub ButtonSubmit_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles ButtonSubmit.Click

        'Initialize memory values all to 00
        For MemCounter = 0 To 256
            Memory.Add("00000000")
        Next

        For MemCounter = 0 To 256
            Memory(MemCounter) = "00000000"
        Next

        Dim filePath As String
        filePath = "test.pl"

        If My.Computer.FileSystem.FileExists(filePath) = False Then
'Verify that the perl file exists.
            MsgBox("File Not Found: " & filePath, MsgBoxStyle.Critical +
MsgBoxStyle.ApplicationModal, "Error")
            Environment.Exit(0)
        Else
        End If

        Dim objFile As New System.IO.StreamWriter("Assembly.txt")
'File to save original code
        Dim intCounter As Long = TextBoxAssemblyCode.Lines.Count
        For intCounter = 0 To TextBoxAssemblyCode.Lines.Count - 1
'For loop to write each line in TextBoxAssemblyCode to file
            objFile.WriteLine(TextBoxAssemblyCode.Lines(intCounter).ToString)
        Next intCounter
        objFile.Close()
```

```
        Shell("perl " + filePath, AppWinStyle.Hide, True)
'Run perl file that does the actual compiling

        Dim objFile1 As New System.IO.StreamReader("Output.txt")        'File
to read output from perl file
        Dim strContents As String
        strContents = objFile1.ReadToEnd()                              'Read
contents of text file, save in variable strContent
        TextBoxMachineCode.Text = strContents
'Display in TextBoxMachineCode
        objFile1.Close()
        objFile1.Dispose()

        Dim ErrorBox As New System.IO.StreamReader("Error.txt")     'Read
error contents
        Dim ErrorContents As String
        ErrorContents = ErrorBox.ReadToEnd()                        'Read
contents of text file, save in variable ErrorBox
        TextBoxError.Text = ErrorContents                           'Display
in TextBoxError
        ErrorBox.Close()
        ErrorBox.Dispose()

        My.Computer.FileSystem.DeleteFile("Assembly.txt")           'Delete
the text files after use
        My.Computer.FileSystem.DeleteFile("Output.txt")
        My.Computer.FileSystem.DeleteFile("Error.txt")



        'Hide Transmit button if code contains errors
        If Not TextBoxError.Text = "Code OK. Proceed with transmission." And
Not TextBoxMachineCode.Text Is Nothing Then
            ButtonTransmit.Enabled = False
        Else
            ButtonTransmit.Enabled = True
        End If

        'Set counter to 0 everytime assembly code is compiled
        TransmitCounter = 0
        TextBoxCounter.Text = "0"
        LabelTotalLine.Text = TextBoxMachineCode.Lines.Count - 1

        'Set all textboxes in EduCPU View to empty
        FormUCView.TextBoxUCInstructions.Text = ""
        FormUCView.TextBoxShift.Text = ""
        FormUCView.TextBoxRegRT.Text = ""
        FormUCView.TextBoxRegRS.Text = ""
        FormUCView.TextBoxRegRDChosen.Text = ""
        FormUCView.TextBoxRegRD.Text = ""
        FormUCView.TextBoxOpcode.Text = ""
        FormUCView.TextBoxImmExt.Text = ""
        FormUCView.TextBoxImm.Text = ""
        FormUCView.TextBoxOverflow.Text = ""
        FormUCView.TextBoxData1.Text = ""
        FormUCView.TextBoxData2.Text = ""
        FormUCView.TextBoxALUInput2.Text = ""
        FormUCView.TextBoxALUResult.Text = ""
        FormUCView.TextBoxZero.Text = ""
        FormUCView.TextBoxNegative.Text = ""
        FormUCView.TextBoxMemData.Text = ""
        FormUCView.TextBoxShifted.Text = ""
```

```
        FormUCView.TextBoxLeft.Text = ""
        FormUCView.TextBoxRight.Text = ""
        FormUCView.TextBoxDataWrite.Text = ""
        FormUCView.TextBoxImmChosen.Text = ""
        FormUCView.TextBoxCtrlRead.Text = ""
        FormUCView.TextBoxCtrlWrite.Text = ""
        FormUCView.LabelInstruction.Text = ""

        'Clears the values of registers 0-7
        For i = 0 To 7
            Register(i) = "00000000"
        Next

        'Displays the cleared values
        FormRegister.TextBox000.Text = Register(0)
        FormRegister.TextBox001.Text = Register(1)
        FormRegister.TextBox010.Text = Register(2)
        FormRegister.TextBox011.Text = Register(3)
        FormRegister.TextBox100.Text = Register(4)
        FormRegister.TextBox101.Text = Register(5)
        FormRegister.TextBox110.Text = Register(6)
        FormRegister.TextBox111.Text = Register(7)

        FormMemory.ButtonChangeValues.Enabled = True

    End Sub

#End Region

#Region "Transmit Button"

    Private Sub ButtonTransmit_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles ButtonTransmit.Click

        Dim MachineCodeStr As String
        Dim InstructionLines() As String
        Dim InstructionType As String

        MachineCodeStr = TextBoxMachineCode.Text                    'To display
instruction in Microcontroller View
        InstructionLines = Split(MachineCodeStr, vbCrLf)
        FormUCView.TextBoxUCInstructions.Text =
InstructionLines(TransmitCounter)
        TransmitCounter = TransmitCounter + 1                       'Increment
TransmitCounter
        TextBoxCounter.Text = TransmitCounter
        FormUCView.TextBoxPC.Text = TransmitCounter

        '----------------------------
        'Simulation codes starts here
        '----------------------------

        'Display which bits goes where in EduCPU View
        FormUCView.TextBoxOpcode.Text =
FormUCView.TextBoxUCInstructions.Text.Substring(0, 5)
        FormUCView.TextBoxShift.Text =
FormUCView.TextBoxUCInstructions.Text.Substring(14, 2)
        FormUCView.TextBoxRegRS.Text =
FormUCView.TextBoxUCInstructions.Text.Substring(5, 3)
        FormUCView.TextBoxRegRT.Text =
FormUCView.TextBoxUCInstructions.Text.Substring(8, 3)
        FormUCView.TextBoxRegRD.Text =
```

```
FormUCView.TextBoxUCInstructions.Text.Substring(11, 3)
        FormUCView.TextBoxImm.Text =
FormUCView.TextBoxUCInstructions.Text.Substring(11, 5)
        FormUCView.TextBoxJumpAddr.Text =
FormUCView.TextBoxUCInstructions.Text.Substring(8, 8)
        FormUCView.TextBoxImmExt.Text = "000" +
FormUCView.TextBoxUCInstructions.Text.Substring(11, 5)
        FormUCView.TextBoxImmChosen.Text = FormUCView.TextBoxImmExt.Text
        '-------------------------------------------

        'If R-Type instructions..
        If FormUCView.TextBoxUCInstructions.Text.Substring(0, 5) = "00001" Or
            FormUCView.TextBoxUCInstructions.Text.Substring(0, 5) = "00011" Or
            FormUCView.TextBoxUCInstructions.Text.Substring(0, 5) = "00100" Or
            FormUCView.TextBoxUCInstructions.Text.Substring(0, 5) = "00110" Or
            FormUCView.TextBoxUCInstructions.Text.Substring(0, 5) = "01000" Or
            FormUCView.TextBoxUCInstructions.Text.Substring(0, 5) = "01010" Or
            FormUCView.TextBoxUCInstructions.Text.Substring(0, 5) = "01011" Or
            FormUCView.TextBoxUCInstructions.Text.Substring(0, 5) = "01110" Or
            FormUCView.TextBoxUCInstructions.Text.Substring(0, 5) = "01111"
Then

            InstructionType = "R"
            FormUCView.TextBoxRegRDChosen.Text =
FormUCView.TextBoxUCInstructions.Text.Substring(11, 3)
        Else
            InstructionType = "I"
            FormUCView.TextBoxRegRDChosen.Text =
FormUCView.TextBoxUCInstructions.Text.Substring(8, 3)
        End If
        '-------------------------------------------

        'Displays data running in EduCPU View
        FormUCView.TextBoxData1.Text =
Register(CDeci(FormUCView.TextBoxRegRS.Text))
        FormUCView.TextBoxData2.Text =
Register(CDeci(FormUCView.TextBoxRegRT.Text))
        FormUCView.TextBoxMemData.Text = "DontCare"
        FormUCView.TextBoxShifted.Text =
Register(CDeci(FormUCView.TextBoxRegRS.Text))
        FormUCView.TextBoxCtrlRead.Text = "0"
        FormUCView.TextBoxCtrlWrite.Text = "0"
        '-------------------------------------

        FormUCView.TextBoxMux3.Text = "1"
        FormUCView.TextBoxMux4.Text = "x"
        FormUCView.TextBoxMux5.Text = "1"

        'Specify which instruction does what
        If FormUCView.TextBoxOpcode.Text = "00010" Then 'addi
            FormUCView.LabelInstruction.Text = "addi"
            Register(CDeci(FormUCView.TextBoxRegRDChosen.Text)) =
CBin8(CDeci(Register(CDeci(FormUCView.TextBoxRegRS.Text))) +
CDeci(FormUCView.TextBoxImm.Text))
            FormUCView.TextBoxALUResult.Text =
Register(CDeci(FormUCView.TextBoxRegRDChosen.Text))
            FormUCView.TextBoxDataWrite.Text =
FormUCView.TextBoxALUResult.Text
            FormUCView.TextBoxMux2.Text = "0"
        ElseIf FormUCView.TextBoxOpcode.Text = "00001" Then 'add
            FormUCView.LabelInstruction.Text = "add"
            Register(CDeci(FormUCView.TextBoxRegRDChosen.Text)) =
```

```
CBin8(CDeci(Register(CDeci(FormUCView.TextBoxRegRS.Text))) +
CDeci(Register(CDeci(FormUCView.TextBoxRegRT.Text))))
            FormUCView.TextBoxALUResult.Text =
Register(CDeci(FormUCView.TextBoxRegRDChosen.Text))
            FormUCView.TextBoxDataWrite.Text =
FormUCView.TextBoxALUResult.Text
            FormUCView.TextBoxMux2.Text = "1"
        ElseIf FormUCView.TextBoxOpcode.Text = "00011" Then 'sub
            FormUCView.LabelInstruction.Text = "sub"
            Register(CDeci(FormUCView.TextBoxRegRDChosen.Text)) =
CBin8(CDeci(Register(CDeci(FormUCView.TextBoxRegRS.Text))) -
CDeci(Register(CDeci(FormUCView.TextBoxRegRT.Text))))
            FormUCView.TextBoxALUResult.Text =
Register(CDeci(FormUCView.TextBoxRegRDChosen.Text))
            FormUCView.TextBoxDataWrite.Text =
FormUCView.TextBoxALUResult.Text
            FormUCView.TextBoxMux2.Text = "1"
        ElseIf FormUCView.TextBoxOpcode.Text = "00100" Then 'and
            FormUCView.LabelInstruction.Text = "and"
            Register(CDeci(FormUCView.TextBoxRegRDChosen.Text)) =
CBin8(CDeci(Register(CDeci(FormUCView.TextBoxRegRS.Text))) And
CDeci(Register(CDeci(FormUCView.TextBoxRegRT.Text))))
            FormUCView.TextBoxALUResult.Text =
Register(CDeci(FormUCView.TextBoxRegRDChosen.Text))
            FormUCView.TextBoxDataWrite.Text =
FormUCView.TextBoxALUResult.Text
            FormUCView.TextBoxMux2.Text = "1"
        ElseIf FormUCView.TextBoxOpcode.Text = "00101" Then 'andi
            FormUCView.LabelInstruction.Text = "andi"
            Register(CDeci(FormUCView.TextBoxRegRDChosen.Text)) =
CBin8(CDeci(Register(CDeci(FormUCView.TextBoxRegRS.Text))) And
CDeci(FormUCView.TextBoxImm.Text))
            FormUCView.TextBoxALUResult.Text =
Register(CDeci(FormUCView.TextBoxRegRDChosen.Text))
            FormUCView.TextBoxDataWrite.Text =
FormUCView.TextBoxALUResult.Text
            FormUCView.TextBoxMux2.Text = "0"
        ElseIf FormUCView.TextBoxOpcode.Text = "00110" Then 'or
            FormUCView.LabelInstruction.Text = "or"
            Register(CDeci(FormUCView.TextBoxRegRDChosen.Text)) =
CBin8(CDeci(Register(CDeci(FormUCView.TextBoxRegRS.Text))) Or
CDeci(Register(CDeci(FormUCView.TextBoxRegRT.Text))))
            FormUCView.TextBoxALUResult.Text =
Register(CDeci(FormUCView.TextBoxRegRDChosen.Text))
            FormUCView.TextBoxDataWrite.Text =
FormUCView.TextBoxALUResult.Text
            FormUCView.TextBoxMux2.Text = "1"
        ElseIf FormUCView.TextBoxOpcode.Text = "00111" Then 'ori
            FormUCView.LabelInstruction.Text = "ori"
            Register(CDeci(FormUCView.TextBoxRegRDChosen.Text)) =
CBin8(CDeci(Register(CDeci(FormUCView.TextBoxRegRS.Text))) Or
CDeci(FormUCView.TextBoxImm.Text))
            FormUCView.TextBoxALUResult.Text =
Register(CDeci(FormUCView.TextBoxRegRDChosen.Text))
            FormUCView.TextBoxDataWrite.Text =
FormUCView.TextBoxALUResult.Text
            FormUCView.TextBoxMux2.Text = "0"
        ElseIf FormUCView.TextBoxOpcode.Text = "01000" Then 'xor
            FormUCView.LabelInstruction.Text = "xor"
            Register(CDeci(FormUCView.TextBoxRegRDChosen.Text)) =
CBin8(CDeci(Register(CDeci(FormUCView.TextBoxRegRS.Text))) Xor
CDeci(Register(CDeci(FormUCView.TextBoxRegRT.Text))))
```

```
            FormUCView.TextBoxALUResult.Text =
Register(CDeci(FormUCView.TextBoxRegRDChosen.Text))
            FormUCView.TextBoxDataWrite.Text =
FormUCView.TextBoxALUResult.Text
            FormUCView.TextBoxMux2.Text = "1"
        ElseIf FormUCView.TextBoxOpcode.Text = "01001" Then 'xori
            FormUCView.LabelInstruction.Text = "xori"
            Register(CDeci(FormUCView.TextBoxRegRDChosen.Text)) =
CBin8(CDeci(Register(CDeci(FormUCView.TextBoxRegRS.Text))) Xor
CDeci(FormUCView.TextBoxImm.Text))
            FormUCView.TextBoxALUResult.Text =
Register(CDeci(FormUCView.TextBoxRegRDChosen.Text))
            FormUCView.TextBoxDataWrite.Text =
FormUCView.TextBoxALUResult.Text
            FormUCView.TextBoxMux2.Text = "0"
        ElseIf FormUCView.TextBoxOpcode.Text = "01010" Then 'sll
            FormUCView.LabelInstruction.Text = "sll"
            Register(CDeci(FormUCView.TextBoxRegRDChosen.Text)) =
Register(CDeci(FormUCView.TextBoxRegRS.Text)).Substring(CDeci(FormUCView.TextB
oxShift.Text), 8 - CDeci(FormUCView.TextBoxShift.Text)).PadRight(8, "0")
            FormUCView.TextBoxALUResult.Text = "DontCare"
            FormUCView.TextBoxShifted.Text =
Register(CDeci(FormUCView.TextBoxRegRDChosen.Text))
            ShiftLeft = "1"
            ShiftRight = "0"
            FormUCView.TextBoxDataWrite.Text = FormUCView.TextBoxShifted.Text
            FormUCView.TextBoxMux2.Text = "1"
            FormUCView.TextBoxMux3.Text = "0"
            FormUCView.TextBoxMux4.Text = "0"
        ElseIf FormUCView.TextBoxOpcode.Text = "01011" Then 'srl
            FormUCView.LabelInstruction.Text = "srl"
            Register(CDeci(FormUCView.TextBoxRegRDChosen.Text)) =
Register(CDeci(FormUCView.TextBoxRegRS.Text)).Substring(0, 8 -
CDeci(FormUCView.TextBoxShift.Text)).PadLeft(8, "0")
            FormUCView.TextBoxALUResult.Text = "DontCare"
            FormUCView.TextBoxShifted.Text =
Register(CDeci(FormUCView.TextBoxRegRDChosen.Text))
            ShiftLeft = "0"
            ShiftRight = "1"
            FormUCView.TextBoxDataWrite.Text = FormUCView.TextBoxShifted.Text
            FormUCView.TextBoxMux2.Text = "1"
            FormUCView.TextBoxMux3.Text = "0"
            FormUCView.TextBoxMux4.Text = "0"
        ElseIf FormUCView.TextBoxOpcode.Text = "01100" Then 'sw
            FormUCView.LabelInstruction.Text = "sw"
            Memory(CDeci(Register(CDeci(FormUCView.TextBoxRegRS.Text))) +
CDeci(FormUCView.TextBoxImm.Text)) =
Register(CDeci(FormUCView.TextBoxRegRT.Text))
            FormUCView.TextBoxALUResult.Text =
CBin8(CDeci(Register(CDeci(FormUCView.TextBoxRegRS.Text))) +
CDeci(FormUCView.TextBoxImm.Text))
            FormUCView.TextBoxDataWrite.Text = "DontCare"
            FormUCView.TextBoxCtrlWrite.Text = "1"
            FormUCView.TextBoxMux2.Text = "0"
            FormUCView.TextBoxMux3.Text = "x"
            FormUCView.TextBoxMux4.Text = "x"
        ElseIf FormUCView.TextBoxOpcode.Text = "01101" Then 'lw
            FormUCView.LabelInstruction.Text = "lw"
            Register(CDeci(FormUCView.TextBoxRegRT.Text)) =
Memory(CDeci(Register(CDeci(FormUCView.TextBoxRegRS.Text))) +
CDeci(FormUCView.TextBoxImm.Text))
            FormUCView.TextBoxMemData.Text =
```

```
Memory(CDeci(Register(CDeci(FormUCView.TextBoxRegRS.Text))) +
CDeci(FormUCView.TextBoxImm.Text))
            FormUCView.TextBoxALUResult.Text =
CBin8(CDeci(Register(CDeci(FormUCView.TextBoxRegRS.Text))) +
CDeci(FormUCView.TextBoxImm.Text))
            FormUCView.TextBoxDataWrite.Text = FormUCView.TextBoxMemData.Text
            FormUCView.TextBoxCtrlRead.Text = "1"
            FormUCView.TextBoxMux2.Text = "0"
            FormUCView.TextBoxMux3.Text = "0"
            FormUCView.TextBoxMux4.Text = "1"
        ElseIf FormUCView.TextBoxOpcode.Text = "01110" Then 'beq
            FormUCView.LabelInstruction.Text = "beq"
            FormUCView.TextBoxALUResult.Text =
CBin8(CDeci(Register(CDeci(FormUCView.TextBoxRegRS.Text))) -
CDeci(Register(CDeci(FormUCView.TextBoxRegRT.Text))))
            If Register(CDeci(FormUCView.TextBoxRegRS.Text)) =
Register(CDeci(FormUCView.TextBoxRegRT.Text)) Then
                TransmitCounter = CDeci(FormUCView.TextBoxImm.Text) - 1
            Else
            End If
            FormUCView.TextBoxDataWrite.Text = "DontCare"
            FormUCView.TextBoxMux2.Text = "1"
            FormUCView.TextBoxMux3.Text = "x"
            FormUCView.TextBoxMux4.Text = "x"
        ElseIf FormUCView.TextBoxOpcode.Text = "01111" Then 'bne
            FormUCView.LabelInstruction.Text = "bne"
            FormUCView.TextBoxALUResult.Text =
CBin8(CDeci(Register(CDeci(FormUCView.TextBoxRegRS.Text))) -
CDeci(Register(CDeci(FormUCView.TextBoxRegRT.Text))))
            If Register(CDeci(FormUCView.TextBoxRegRS.Text)) =
Register(CDeci(FormUCView.TextBoxRegRT.Text)) Then
            Else
                TransmitCounter = CDeci(FormUCView.TextBoxImm.Text) - 1
            End If
            FormUCView.TextBoxDataWrite.Text = "DontCare"
            FormUCView.TextBoxMux2.Text = "1"
            FormUCView.TextBoxMux3.Text = "x"
            FormUCView.TextBoxMux4.Text = "x"
        ElseIf FormUCView.TextBoxOpcode.Text = "10000" Then 'j
            FormUCView.LabelInstruction.Text = "j"
            TransmitCounter = CDeci(FormUCView.TextBoxJumpAddr.Text) - 1
            FormUCView.TextBoxALUResult.Text = "DontCare"
            FormUCView.TextBoxDataWrite.Text =
FormUCView.TextBoxALUResult.Text
            FormUCView.TextBoxMux2.Text = "0"
            FormUCView.TextBoxMux3.Text = "x"
            FormUCView.TextBoxMux4.Text = "x"
            FormUCView.TextBoxMux5.Text = "0"
        End If
        '-----------------------------------

        'Check zero flag
        If FormUCView.TextBoxALUResult.Text = "00000000" Then
            ZeroFlag = 1
        Else
            ZeroFlag = 0
        End If
        '-----------------------------------

        'Displays all the registers and flags values
        FormRegister.TextBox000.Text = Register(0)
        FormRegister.TextBox001.Text = Register(1)
```

```
        FormRegister.TextBox010.Text = Register(2)
        FormRegister.TextBox011.Text = Register(3)
        FormRegister.TextBox100.Text = Register(4)
        FormRegister.TextBox101.Text = Register(5)
        FormRegister.TextBox110.Text = Register(6)
        FormRegister.TextBox111.Text = Register(7)
        FormUCView.TextBoxOverflow.Text = OverflowFlag
        FormUCView.TextBoxZero.Text = ZeroFlag
        FormUCView.TextBoxNegative.Text = NegativeFlag
        FormUCView.TextBoxLeft.Text = ShiftLeft
        FormUCView.TextBoxRight.Text = ShiftRight

        If InstructionType = "R" Then
            FormUCView.TextBoxALUInput2.Text = FormUCView.TextBoxData2.Text
            FormUCView.TextBoxMux1.Text = "0"
        Else
            FormUCView.TextBoxALUInput2.Text =
FormUCView.TextBoxImmChosen.Text
            FormUCView.TextBoxMux1.Text = "1"
        End If
        '-------------------------------------------

        'Reset values
        ShiftLeft = 0
        ShiftRight = 0

        If TransmitCounter >= TextBoxMachineCode.Lines.Count - 1 Or
TransmitCounter < 0 Then
            ButtonTransmit.Enabled = False                          'Disable
the Transmit button
        End If

        'Transmit through serial
        If COMPort.IsOpen Then
            ButtonTransmit.Enabled = False
            COMPort.Write(FormUCView.TextBoxUCInstructions.Text.Substring(0,
2))
            Sleep(220)
            COMPort.Write(FormUCView.TextBoxUCInstructions.Text.Substring(2,
2))
            Sleep(220)
            COMPort.Write(FormUCView.TextBoxUCInstructions.Text.Substring(4,
2))
            Sleep(220)
            COMPort.Write(FormUCView.TextBoxUCInstructions.Text.Substring(6,
2))
            Sleep(220)
            COMPort.Write(FormUCView.TextBoxUCInstructions.Text.Substring(8,
2))
            Sleep(220)
            COMPort.Write(FormUCView.TextBoxUCInstructions.Text.Substring(10,
2))
            Sleep(220)
            COMPort.Write(FormUCView.TextBoxUCInstructions.Text.Substring(12,
2))
            Sleep(220)
            COMPort.Write(FormUCView.TextBoxUCInstructions.Text.Substring(14,
2))
            Sleep(220)
            COMPort.Write(vbCr)
            If TransmitCounter >= TextBoxMachineCode.Lines.Count - 1 Or
TransmitCounter < 0 Then
```

```
                ButtonTransmit.Enabled = False
'Disable the Transmit button
            Else
                ButtonTransmit.Enabled = True
            End If
        End If

    End Sub

#End Region

#Region "Other Forms Buttons"

    Private Sub ButtonRegisters_Click(ByVal sender As System.Object, ByVal e
As System.EventArgs) Handles ButtonRegisters.Click
        FormRegister.Show()
    End Sub

    Private Sub ButtonMemoryView_Click(ByVal sender As System.Object, ByVal e
As System.EventArgs) Handles ButtonMemoryView.Click
        FormMemory.Show()
    End Sub

    Private Sub ButtonViewUC_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles ButtonViewUC.Click
        FormUCView.Show()
    End Sub

    Private Sub ButtonHelp_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles ButtonHelp.Click
        FormHelp.Show()
    End Sub

#End Region

#Region "Functions"
    'converts an integer to binary string
    Public Function CBin(ByVal DecimalNum As Long) As String
        Dim tmp As String
        Dim n As Long
        n = DecimalNum
        tmp = Trim(Str(n Mod 2))
        n = n \ 2
        Do While n <> 0
            tmp = Trim(Str(n Mod 2)) & tmp
            n = n \ 2
        Loop
        CBin = tmp
    End Function

    'converts a binary string to integer
    Public Function CDeci(ByRef s As String) As Integer
        Dim i As Long
        CDeci = 0
        For i = 0 To Len(s) - 1
            CDeci = CDeci + (Mid$(s, Len(s) - i, 1) * 2 ^ i)
        Next i
    End Function

    'converts an integer to 8-bit binary
    Public Function CBin8(ByVal n As Integer) As String
        If n > 255 Then         'If bigger than 8 bit
```

```
            n = n - 256
            OverflowFlag = 1    'Set corresponding flag
            NegativeFlag = 0    'Set corresponding flag
            Dim i As Int64 = Convert.ToInt16(n)
            CBin8 = Convert.ToString(i, 2).PadLeft(8, "0")
        ElseIf n < 0 Then       'If negative
            Dim i As Int64 = Convert.ToInt16(n)
            Dim TempBin As String
            TempBin = Convert.ToString(i, 2).PadLeft(8, "0")
            CBin8 = TempBin.Substring(TempBin.Length - 8, 8)
            OverflowFlag = 1    'Set corresponding flag
            NegativeFlag = 1    'Set corresponding flag
        Else
            Dim i As Int64 = Convert.ToInt16(n)
            CBin8 = Convert.ToString(i, 2).PadLeft(8, "0")
            OverflowFlag = 0    'Set corresponding flag
            NegativeFlag = 0    'Set corresponding flag
        End If

    End Function

#End Region

#Region "Serial Ports Settings"

    Private Sub FormMainView_Load(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles MyBase.Load
        For Each COMString As String In My.Computer.Ports.SerialPortNames '
Load all available COM ports.
            ComboBoxCOMPort.Items.Add(COMString)
        Next
        ComboBoxCOMPort.Sorted = True
    End Sub

    Private Sub ComboBoxCOMPort_SelectedIndexChanged(ByVal sender As
System.Object, ByVal e As System.EventArgs) Handles
ComboBoxCOMPort.SelectedIndexChanged
        COMLamp.BackColor = Color.Gray
        COMLamp.Text = "Port closed"
        'DTRLamp.BackColor = Color.Gray
        If COMPort.IsOpen Then
            COMPort.RtsEnable = False
            COMPort.DtrEnable = False
            ClosePort()
            Application.DoEvents()
            Sleep(200)                           ' Wait 0.2 second for port to
close as this does not happen immediately.
        End If
        COMPort.PortName = ComboBoxCOMPort.Text
        COMPort.BaudRate = 9600                  ' Default for Max-i: 19200 bit/s,
8 data bits, no parity, 1 stop bit
        COMPort.WriteTimeout = 2000              ' Max time to wait for CTS = 2
sec.

        Try
            COMPort.Open()
        Catch ex As Exception
            MsgBox(ex.Message)
        End Try

        ComboBoxBaudRate.Text = COMPort.BaudRate.ToString
        ComboBoxDataBits.Text = COMPort.DataBits.ToString
```

```
        ComboBoxParity.Text = COMPort.Parity.ToString
        ComboBoxFlowControl.Text = COMPort.Handshake.ToString

        If COMPort.IsOpen Then
            ButtonClosePort.Visible = True
            COMPort.RtsEnable = True
            COMLamp.Text = ComboBoxCOMPort.Text + " opened"
            COMLamp.BackColor = Color.LightGreen
            COMPort.DtrEnable = True
            ' DTRLamp.BackColor = Color.LightGreen
        End If
    End Sub

    Private Sub ClosePort()
        If COMPort.IsOpen Then COMPort.Close()
    End Sub

    Private Sub ComboBoxDataBits_SelectedIndexChanged(ByVal sender As
System.Object, ByVal e As System.EventArgs) Handles
ComboBoxDataBits.SelectedIndexChanged
        COMPort.DataBits = CInt(ComboBoxDataBits.Text)
    End Sub

    Private Sub ComboBoxBaudRate_SelectedIndexChanged(ByVal sender As
System.Object, ByVal e As System.EventArgs) Handles
ComboBoxBaudRate.SelectedIndexChanged
        COMPort.BaudRate = CInt(ComboBoxBaudRate.Text)
    End Sub

    Private Sub ComboBoxParity_SelectedIndexChanged(ByVal sender As
System.Object, ByVal e As System.EventArgs) Handles
ComboBoxParity.SelectedIndexChanged
        COMPort.Parity = CType([Enum].Parse(GetType(Parity),
ComboBoxParity.Text), Parity)
    End Sub

    Private Sub ComboBoxFlowControl_SelectedIndexChanged(ByVal sender As
System.Object, ByVal e As System.EventArgs) Handles
ComboBoxFlowControl.SelectedIndexChanged
        COMPort.Handshake = CType([Enum].Parse(GetType(Handshake),
ComboBoxFlowControl.Text), Handshake)
    End Sub

    Private Sub ButtonClosePort_Click(ByVal sender As System.Object, ByVal e
As System.EventArgs) Handles ButtonClosePort.Click
        ClosePort()
        COMLamp.BackColor = Color.Gray
        COMLamp.Text = "Port closed"
        ComboBoxBaudRate.Text = ""
        ComboBoxDataBits.Text = ""
        ComboBoxParity.Text = ""
        ComboBoxFlowControl.Text = ""
        ComboBoxCOMPort.Text = ""
        ButtonClosePort.Visible = False
    End Sub

#End Region


End Class
```